

ND-A141 083

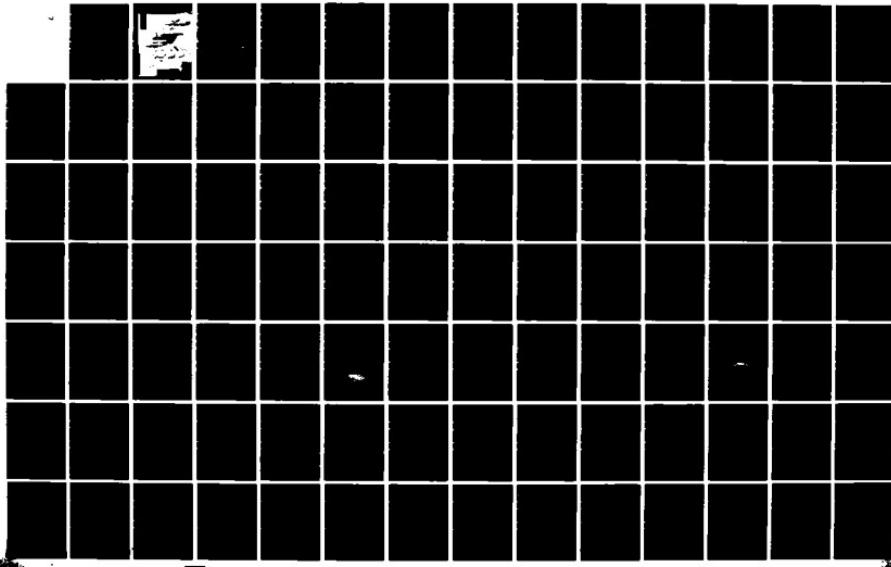
MULTIPROCESSOR Z-BUFFER ARCHITECTURE FOR HIGH-SPEED
HIGH COMPLEXITY COMPUTER IMAGE GENERATION(U) BOEING
AEROSPACE CO SEATTLE WA DEC 83 MDA903-82-C-0101

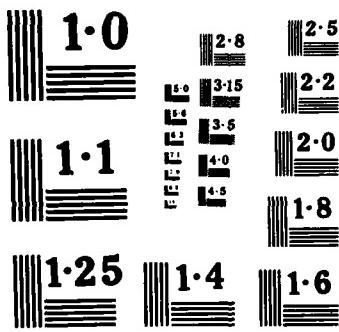
14

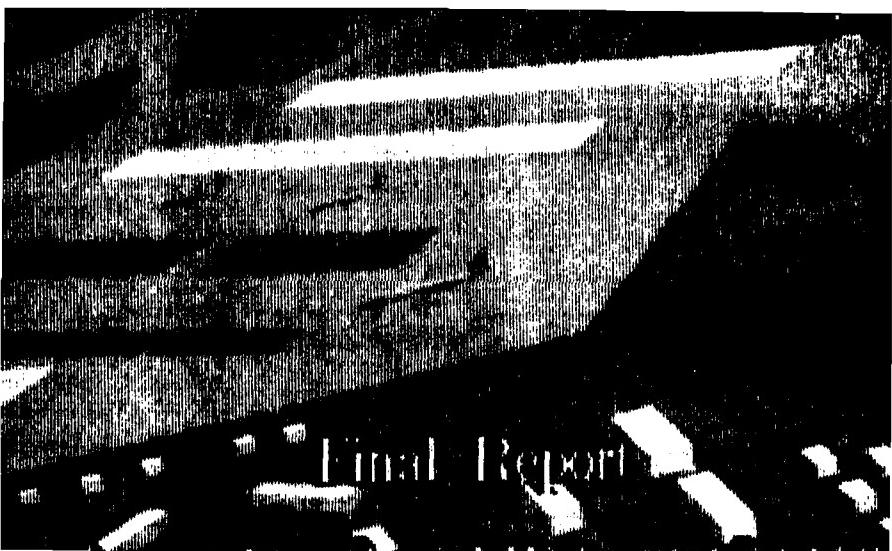
UNCLASSIFIED

F/G 9/2

NL







**A Multiprocessor Z-Buffer Architecture
for High-Speed, High-Complexity
Computer Image Generation**

December 1983

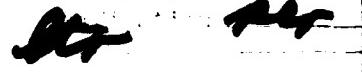
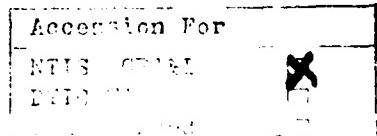
**Prepared for:
Information Processing Techniques Office
Defense Advanced Research Projects Agency
Washington, D.C.**

Final Report, Contract No: MDA903-82-C-0101

**Advanced Computer Graphics Technology
Boeing Aerospace Company
Seattle, Washington**

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

This document has been approved
for public release and sale; its
distribution is unlimited.



BTIC Reproduct
by the Black Card

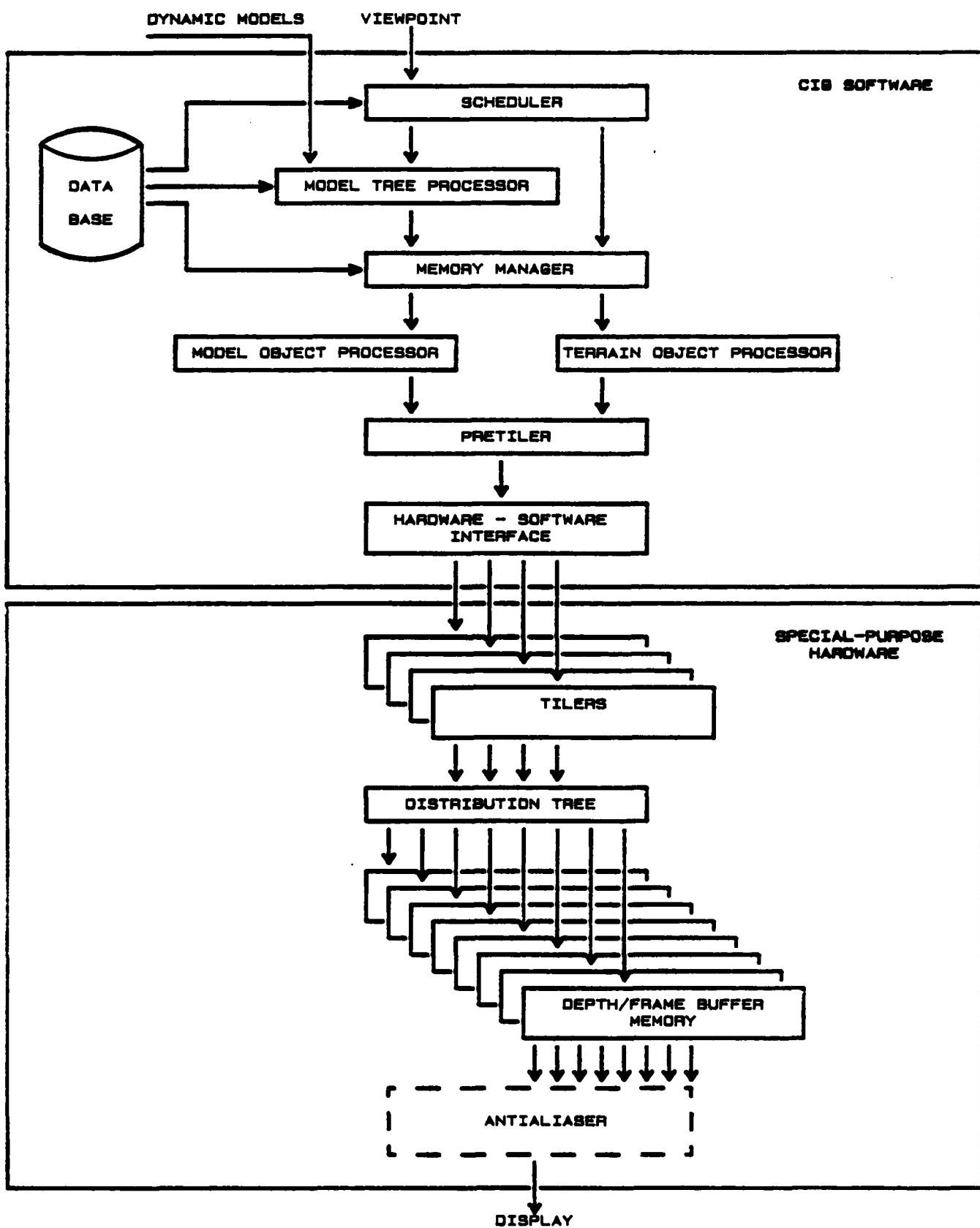
ABSTRACT

This report describes the results of the second phase of a three-phase program, sponsored by the Defense Advanced Research Projects Agency (DARPA), to develop an innovative, high-complexity computer image generation (CIG) system. The foundation of this CIG system is a multi-processor implementation of the z-buffer hidden-surface algorithm.

The research program accomplished the following:

- Real-time special-purpose hardware to render shaded triangles into a displayable image. It consists of four tilers, a distribution tree, and eight depth/frame buffer memories, as shown in the following diagram.
- A non-real-time software simulation of the front-end of a CIG system. It consists of a hardware-software interface, a pretiler, model and terrain object processors, a memory manager, a model tree processor, and a scheduler.
- Software to construct databases from the Defense Mapping Agency's Digital Terrain Elevation Data tapes and from the output of a general-purpose CAD system (Graftek). and
- Software developed to investigate additional topics in computer graphics, such as fractals, curved surfaces, transparencies, texture mapping, and level-of-detail control.

It was concluded that the proposed architecture is suitable for real-time and near-real-time CIG applications, and takes full advantage of the flexibility offered by the z-buffer algorithm.



Module	Purpose	Comments
Depth/Frame Buffer Module	Performs hidden-surface elimination and stores pixel information.	<ul style="list-style-type: none"> o Total pixel resolution = $512 \times 512 \times 8$ bits o Depth represented as 24-bit, floating-point inverse distance, and color as 8-bit, fixed-point intensity. o Segmented into eight modules. o Each module can process an incoming pixel every $115\text{-}210$ ns, for an aggregate throughput of greater than 4×10^7 pixels/second.
Tiler	Computes screen location, depth, and color for each pixel covered by the triangle.	<ul style="list-style-type: none"> o Utilize incremental algorithm which addresses a number of problems inherent in z-buffer systems. o Each tiler can accept a triangle every 10 microseconds for an aggregate triangle throughput of 4×10^5 triangles/second. o Each tiler can output a pixel every 100 ns for an aggregate pixel throughput of 4×10^7 pixels/second.
Distribution Tree	Interconnects the four tilers to the eight memory modules.	<ul style="list-style-type: none"> o Allows data to be passed from any tiler to any memory module. o Maintains a uniform distribution of data among the memory modules. o Resolves contention resulting from two tilers simultaneously accessing the same memory module. o Accepts pixel data at each input every 100 ns, yielding an aggregate throughput of 4×10^7 pixels/second.
Antialiaser	Reduces undesirable visual effects caused by finite pixel resolution.	<ul style="list-style-type: none"> o Utilizes 4:1 oversampling and a 3×3 pyramid filter. o Was not built since the above antialiasing technique was found to give unacceptable results.

Module	Purpose	Comments
Hardware-Software Interface	Controls synchronization and outputs triangles to tilers.	
Pretiler	Clips, projects, and parameterizes a triangle in preparation for tiling.	<ul style="list-style-type: none"> o Reduction in complexity results from performing screen-boundary clipping in 2-space rather than 3-space. o Unnecessary projection calculations for adjacent triangles are avoided by storing results and retrieving them if needed.
Model Object Processor	Transforms, shades, and triangulates arbitrary polygonal objects.	<ul style="list-style-type: none"> o Allows curve or face shading, and vertex or polygon coloring. o Unnecessary transformation calculations for adjacent triangles are avoided by storing results and retrieving them if needed. o Eliminates back-facing triangles.
Terrain Object Processor	Transforms, shades, and triangulates a rectangular grid of topographic data.	<ul style="list-style-type: none"> o Utilizes regular structure of grid to drastically reduce computations. o Unnecessary transformation calculations for adjacent triangles are avoided by storing results and retrieving them if needed. o Grid spacing is arbitrary. o Eliminates back-facing triangles.

Module	Purpose	Comments
Memory Manager	Reads necessary data from disk-resident database and stores it in memory.	
Model Tree Processor	Traverses a tree structure defining a model.	<ul style="list-style-type: none"> o Processes dynamic or static models. o Allows moving or detachable parts. o Performs field-of-view and level-of-detail processing.
Scheduler	Traverses a tree structure defining the terrain and static models placed on it.	<ul style="list-style-type: none"> o Preserves database flexibility by treating all nodes identically. o Performs field-of-view and level-of-detail processing.

Summary of CIG Software (Concluded)

ACKNOWLEDGEMENTS

It is with great personal pride and pleasure that I take this opportunity to acknowledge the contributions made to the success of the Syntactic Visual Image Generation program.

First and foremost, I would like to thank Dr. Craig Fields and Lt. Col. Jack Thorpe, of the Defense Advanced Research Projects Agency, for their unflagging energy, guidance, encouragement and support. I want to particularly thank Lt. Col. Thorpe, whose extraordinary management skills and personal enthusiasm made accomplishing this program possible.

Mr. Herb Hoy, of NASA/AMES, acted in the vital role of the contract officer's technical representative. He was responsible for providing insightful feedback to the Boeing research team and monitoring the project on behalf of the contract committee. This committee, which provided support essential to the success of this effort, was composed of the following individuals: George Lukes, Col. William Harmon, Hank King, Dr. Carl Verhey, Dr. Joe Golden, Lt. Col. Hank Dorr, and Col. Dick Webb.

For daring to imagine the "impossible," and having the courage, personal will, and creativity to make their dream of a "graphics multiprocessor" real, I want to thank Robert Rife, Earl Wilson, Andrew Johnston, Steve Black, and Dr. Chris Allen.

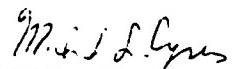
For excellence in design, execution, and administration, in simultaneously solving the key theoretical problems and translating those solutions into hundreds of thousands of lines of software and operational hardware: Rick Bess, Dr. Francis Cheng, Carl Christofferson, Sharon Drake, Randy Hellman, Sandra Hoyt, Mark Illing, Rich Johnston, Philip Keller, Dr. Bob Markot, John Methot, Dave Moerdyk, Mike Nelson, Ann Rupley, George Sabik, Pat Salzetti, Barry Shaw, Mark Snitily, Brian Soderberg, Jody Wahlgren, Marcia Watkins, Dr. Pete Wever, and Sonja Willanger.

For creating and maintaining a superior, productive work environment, and the vital management support needed to keep the program on track, I want to thank Robert Hager, Ivan Stampalia, and Kegam Budak.

For exceptional, contract and materiel support, Tommie Overcast, Jean Testu,
Dave Ingram, and Diane Weller.

And finally, a special word of thanks and gratitude to our families, for their
sacrifice and moral support during the past two years.

Respectfully submitted,



Michael L. Cyrus
Boeing Program Manager

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
	GLOSSARY	xv
1.0	INTRODUCTION	1
2.0	Z-BUFFER ALGORITHM	5
3.0	SYSTEMS DESIGN	13
	3.1 MULTIPROCESSOR ARCHITECTURE	14
	3.2 COMPROMISES	18
	3.3 SYSTEM OVERVIEW	20
	3.4 HARDWARE OVERVIEW	24
	3.5 SOFTWARE OVERVIEW	27
4.0	HARDWARE DESIGN	31
	4.1 DEPTH/FRAME BUFFER MEMORY	32
	4.2 TILERS	38
	4.3 DISTRIBUTION TREE	71
	4.4 ANTIALIASING	95
5.0	SOFTWARE DESIGN	103
	5.1 DATABASE CONSTRUCTION	104
	5.2 HARDWARE-SOFTWARE INTERFACE	123
	5.3 PRETILER	131
	5.4 MODEL OBJECT PROCESSOR (MOP)	137
	5.5 TERRAIN OBJECT PROCESSOR (TOP)	145
	5.6 MEMORY MANAGER	164
	5.7 MODEL TREE PROCESSOR	167
	5.8 SCHEDULER	171

TABLE OF CONTENTS (concluded)

<u>Section</u>		<u>Page</u>
6.0	GRAPHICS ALGORITHM RESEARCH	175
6.1	FRACTALS	176
6.2	CURVED SURFACES	182
6.3	TRANSPARENCY	193
6.4	TEXTURE AND COLOR MAPPING	198
6.5	LEVEL-OF-DETAIL CONTROL	211
7.0	PERFORMANCE EVALUATION	220
7.1	HARDWARE PERFORMANCE	221
7.2	SOFTWARE PERFORMANCE	242
8.0	CONCLUSION	253
8.1	PROBLEM AREAS	254
8.2	ACCOMPLISHMENTS	255
8.3	FUTURE DEVELOPMENT	256
9.0	REFERENCES	257
	APPENDIX A - HISTORICAL BACKGROUND	267
A.1	GRAPHICS RESEARCH	268
A.2	FLIGHT SIMULATION	270
A.3	TECHNOLOGY TRENDS	274
	PLATES	276

LIST OF FIGURES

<u>Figure No.</u>	<u>Title</u>	<u>Page</u>
1.	Definition of Z-Coordinate (Depth)	6
2.	Cluster Priority	7
3.	Face Priority	8
4.	Traumatic Geometric Constructs	12
5.	Queues Used to Smooth Data Flow	15
6.	No-Scrambling Illustration	17
7.	Scrambling Illustration	17
8.	DARPA System	20
9.	Architecture Research System (ARS)	21
10.	Special-Purpose Hardware	25
11.	CIG Software	28
12.	Depth Exponent Compare	36
13.	Reprojection	40
14.	Projected Color and Linearity	45
15.	Dot Product Test	48
16.	Undersampling and Oversampling	50
17.	"Poking Through" Effects	51
18.	Sampling Paths	52
19.	Triangle Variables	54
20.	Intelligent Tiling Algorithm	61
21.	Tiler Functional Blocks	64
22.	HSD Interface	65
23.	Tiling Machine Setup	67
24.	Tiling Machine	68
25.	Tile Accumulate	69
26.	A 1x8 Sorting Machine	77
27.	A 2x8 Sorting Machine	79
28.	A 4x8 Sorting Machine	80
29.	Cell Queue Length ~ Tree Level	84
30.	Tree Contention by Queuing Model	85
31.	Type A-cell Functional Diagram	87
32.	Type B-cell Functional Diagram	90

LIST OF FIGURES (continued)

<u>Figure No.</u>	<u>Title</u>	<u>Page</u>
33.	Aliasing Effects - Discrete Sampling	97
34.	Aliasing Effects - Stairstepping	97
35.	Aliasing Effects - Crawling	98
36.	Aliasing Effects - Scintillation	98
37.	Effects of Antialiasing	99
38.	4x Oversampling and Filtering	100
39.	Level One Data Combination	100
40.	Logical Input to the Antialiaser	101
41.	Database File Network	106
42.	Sample Tree Structure	108
43.	Merging Gaming Areas for Continuous Flight Path	109
44.	DMA Raw Terrain Data Management	112
45.	Raw Gaming Area Array (RAWGA)	113
46.	Sample Gaming Area and TED Quadtree	114
47.	Node Variable Development	116
48.	Delta-Z Array	117
49.	Required Raw Data for Vertex Normals	120
50.	Hardware-Software Interface Communication Channels	125
51.	Data Format for One Triangle	127
52.	Corner Definition	127
53.	Buffer/Tiler Interaction	130
54.	Spatial Orientation of Viewpoint and Screen Space Coordinates	132
55.	Hither and Yon Clipping	133
56.	Hither Clipping	134
57.	Clipping to Screen Space	135
58.	Terrain Object Processor Functional Flow	148
59.	Terrain Delta-Z Grid	150
60.	DMA Terrain Grid	151
61.	Six-sided Box with Three Front Faces and Three Back Faces	156
62.	Terrain Face Construction	157
63.	Mount Rainier - Gouraud Shaded Terrain	161
64.	Mount Rainier - Texture Mapped Terrain	162

LIST OF FIGURES (continued)

<u>Figure No.</u>	<u>Title</u>	<u>Page</u>
65.	Model Object Tree Structure	169
66.	Mount Rainier Enhanced by Fractals	177
67.	Triangle Subdivision	178
68.	B-Spline Surface and Control Point Grid	184
69.	Two Closed B-Spline Curves	184
70.	Subdivision on Bezier Curves	187
71.	A Bicubic Bezier Patch as Rendered by the Subdivision Algorithm and its Original Control Grid	189
72.	Tests for Flatness	190
73.	Teapot Modeled with Twenty-eight Bicubic Bezier Patches	192
74.	Projection of Texture Map	200
75.	Memory Arrangement of Color Mip Maps	203
76.	Calculating Mip LOD	205
77.	Extracting Final Color Value from a Mip Map	206
78.	Projection of Aerial Photo Data to Texture	209
79.	Projection of Texture Maps to Model Faces	210
80.	Triangle Generation of Downsampled Terrain Data	213
81.	Model Tree with LOD	214
82.	Crack-of-the-Earth Due to Data Reduction	216
83.	LOD Boundary Range Values (R_j)	217
84.	Memory Module Cycle Utilization	225
85.	Average Memory Cycle Lengths for Four Design Alternatives	226
86.	Tiler Transport Delay	227
87.	Effect of Triangle Size on Tiler Throughput Rates	229
88.	Tiling Machine Setup Stage Performance for Oversample Mode	234
89.	Tiling Machine Setup Stage Performance for Undersample Mode	235
90.	Tiling Efficiency for Undersample and Oversample Mode	236
91.	Maximum Queue Lengths Required for Sequential, Interleave, and Scramble Modes at Varying Data Rates	239
92.	Distribution of Memory Accesses for Sequential, Interleave, and Scramble Modes	240

LIST OF FIGURES (continued)

<u>Figure No.</u>	<u>Title</u>	<u>Page</u>
93.	Relative Time Per Face for Major Software Tasks	244
94.	Percentage of Time Per Frame for Major Tasks (Typical Frame)	245
95.	Flow of Faces in Field of View to Tiler (Typical Frame)	247
96.	Use of Sixteen Subimages to Form a Single Large Image	248
97.	Distribution of Faces in FOV by Level of Detail	251
98.	Increases in Device Complexity through Time	274

LIST OF TABLES

<u>Table No.</u>	<u>Title</u>	<u>Page</u>
1.	Relative Computational Costs of Different Hidden-Surface Algorithms	11
2.	Depth Resolution ΔU for Different Representations of Depth	42
3.	Triangle Variable Definition	55
4.	Tiling Machine Setup Variables	56
5.	Tiling Machine Setup Calculations	57
6.	Tiling Machine Input Variables	59
7.	Tiling Machine Internal Variables	60
8.	Tiler Variable Precision and Format	63
9.	A-cell Controller Algorithm	89
10.	B-cell Controller Algorithm	91
11.	Vertex Normal Derivation	118
12.	Shading and Coloring Options	141
13.	Model Object Processor Algorithm	142
14.	Per Vertex Coordinate Transformation Cost	155
15.	Cross Product Computation for Two Faces	159
16.	Model Tree Processor Algorithm	170
17.	Two-Dimensional Fractal Generation Algorithm	179
18.	Level of Detail Range Algorithm	218
19.	Memory Module Cycles	222
20.	Memory Statistics	224
21.	Tiling Machine Setup Average Processing Time Per Triangle	228
22.	Tiler Statistics	230
23.	Sampling Efficiencies for Undersample and Oversample Modes	231
24.	Tiling Efficiencies for Undersample and Oversample Modes	232
25.	Scenario for Software Performance Measurement	243
26.	Screen Space Projection - Shared Vertices (Typical Frame)	249
27.	Speed Improvement Using Vertex Buffer for Projection (Typical Frame)	249
28.	Comparison of Model and Terrain Processing Algorithms When Processing an Equal Number of Faces	250
29.	LOD Control by Face Count and Range (Typical Frame)	251

GLOSSARY

A,B,C Space -
See Model Space.

Active Edge Table -
In a scan-line hidden-surface algorithm, a table that contains all the polygonal edges that intersect the scanline.

AET -
See Active Edge Table.

Aliasing -
A variety of distracting visual effects over space or time, such as jagged edges and scintillation, caused by the finite pixel resolution of a display system.

Antialiasing -
Techniques used to minimize aliasing effects.

Architecture Research System -
A system of software and special-purpose hardware to research a CIG system architecture based on a depth-buffer algorithm.

ARS -
See Architecture Research System.

Attribute data -
Data that defines characteristics of the surface of an object, such as color and reflectivity.

B-spline -
Piecewise parametric polynomials that combine control points, parameter space values called knots, and a set of blending functions to define a surface.

Back face -
A polygon facing away from the viewpoint, such as the back side of a mountain.

Background bit -
A bit flag that instructs the memory controller to perform an automatic overwrite to memory.

Beta-spline -
An extension to B-splines that redefines continuity at joints to allow specification of tension and bias parameters.

BG -
See Background bit.

Blending functions -
Parametric polynomials that form a mathematical basis for B-splines.

- Buffer -**
Memory used to store large blocks of data.
- CIG -**
See Computer Image Generation.
- Clipping -**
Removing the sections of a polygon that lie outside some specified set of boundaries such as a display screen.
- Cluster Node -**
In a database structure, cluster nodes refer to nodes that do not have data associated with them; they only contain pointers to their children.
- CMOS -**
Abbreviation for Complementary Metal-Oxide Semiconductor. CMOS is a technology for fabricating integrated circuits that uses p- and n-channel FET transistors for its logic. See FET.
- Color Map -**
A variable-size, two-dimensional array of R,G,B color values that can be projected onto terrain or models.
- Color Shading -**
Calculation of intensity values of the R,G,B components at a point on a surface depending on the sun angle.
- Computer Graphics -**
The creation, storage, and manipulation of pictures of real or imaginary objects via a computer.
- Computer Image Generation -**
The process of generating a shaded, three-dimensional, perspective correct scene by computer.
- Control points -**
Points in 3-space that are a component of the B-spline polynomial and can be used to control the shape of the B-spline surface.
- Control/Status Board -**
A processor that was used to control the operation of the laboratory ARS and gather run-time statistics.
- Crack-of-the-earth -**
Holes in terrain models caused by transitioning from one level of detail (LOD) to another.
- CSB -**
See Control/Status Board.
- Curve Shading -**
A method of color shading which causes planar polygons to appear as if they were curved. It is also referred to as Gouraud shading.

- Delta-color -**
The amount of color change from one pixel to the next.
- Delta-depth -**
The amount of depth change from one pixel to the next.
- Depth -**
A measure of the distance between the viewer and any surface. In the Boeing CIG system, depth is the inverse of this distance measured along a vector normal to the screen.
- Depth buffer -**
A buffer that contains depth values for each pixel on the screen. These depth values measure the distance from the viewpoint to the surface represented by the pixel.
- Depth Complexity -**
The average number of distinct surfaces that are penetrated by an infinite line of sight extending from the viewpoint through an arbitrary pixel.
- Depth/Frame Buffer Memories -**
Custom hardware that performs hidden-surface elimination and stores color and depth information for each pixel.
- DFAD -**
See Digital Feature Analysis Data.
- Digital Feature Analysis Data -**
Defense Mapping Agency (DMA) data describing cultural objects and secondary terrain features such as terrain composition.
- Digital Terrain Elevation Data -**
Defense Mapping Agency (DMA) data describing terrain elevations.
- Distribution Tree -**
A network used to interconnect multiple processors and multiple memories. In the architecture research system, the distribution tree distributes data from the four tilers to the eight depth/frame buffer memories.
- DMA -**
Abbreviation for Defense Mapping Agency.
- Double Buffering -**
Data-sharing scheme whereby one task processes data in one buffer while another processes data in the other.
- Downsampling -**
A data sampling scheme that reduces the amount of data while maintaining a good representation to the original data.
- DTED -**
See Digital Terrain Elevation Data.

ECL -

Abbreviation for Emitter Coupled Logic. ECL is a technology for fabricating integrated circuits that uses bipolar digital transistors. It speeds up integrated circuit operation through a more complex design than TTL.

Edge Table -

In a scan-line hidden-surface algorithm, a table containing all non-horizontal polygonal edges, sorted by y-coordinate and slope.

ET -

See Edge Table.

Face -

The surface of a polygon or triangle.

Face Normal -

A vector that is perpendicular to the surface of a polygon.

Face Shading -

A method of color shading that shades the entire face with a single, homogeneous color.

Fail Soft -

The ability of a system to gracefully degrade its performance when failure occurs in parts of the system.

FET -

See Field-effect Transistor.

Field of View -

The volume of space which encompasses all objects that are visible from a specific viewpoint and view angle.

Field-effect Transistor -

A transistor that allows the electrical charge on the wire to flow through the silicon beneath it. A p-channel FET conducts current when the wire is charged negatively and an n-channel FET conducts when the wire is charged positively.

FIFO -

First-In/First-Out memory.

FOV -

See Field of View.

Fractal -

A mathematical model that defines a shape in which increasing detail is revealed with increasing magnification. The structure revealed at each higher level of magnification is similar to the form displayed at lower levels of magnification.

Frame -

A single image or picture on a display.

- Frame Buffer -**
A block of memory that contains the data to produce a single frame.
- Frame-to-frame Coherence -**
Refers to the fact that one frame of a sequence is usually very similar to the one that preceded it.
- Free List -**
In the scheduler, the list of nodes which are no longer needed.
- Geodetic -**
Pertaining to a world coordinate system using latitude and longitude.
- Geometry Data -**
Data used to define the shape of an object.
- Gouraud Shading -**
See Curve Shading.
- Graphics Processor -**
The combination of a frame buffer and a special-purpose processor that is designed to modify the contents of the frame buffer.
- Hardware/Software Interface -**
In the ARS, a module which communicates between the special-purpose hardware and the CIG software.
- Hidden Surface -**
A surface that is not a back face but is hidden from view by another surface.
- High Speed Data -**
General-purpose high-speed parallel device interface used on Gould/SEL computers.
- Hither Plane -**
A plane which defines the field of view (FOV) boundary nearest the viewpoint.
- Host Computer -**
The primary or controlling computer in a multiple-machine system.
- HSD -**
See High Speed Data.
- HSI -**
See Hardware/Software Interface.
- i,j Space -**
See Screen Space.
- INIT -**
See Initialize bit.

Initialize bit -

A bit flag that indicates whether data has been previously written to a specified memory location. It instructs the memory controller to either perform an automatic overwrite to uninitialized memory or initiate a compare cycle.

Knot -

A value in the parameter space over which a B-spline is defined. It locates a juncture of the piecewise polynomials of a B-spline.

Landsat -

Multi-spectral scanner data of the earth's surface gathered by satellite.

Landuse -

Land cover features of terrain, such as glaciers, swamps, and wheat fields.

Leaf Node -

The nodes at the lowest level of a tree structure.

Level of Detail -

The definition of an object at various resolutions to eliminate processing of detail which cannot be distinguished as the object moves further from the viewpoint.

Level V -

High-resolution DFAD database.

LOD -

See Level of Detail.

LSI -

Abbreviation for Large Scale Integration. A description of the number of transistors an integrated circuit contains. LSI circuits are usually designed with 2,000 to 10,000 transistors.

MDD -

Master Dimension Data.

Memory Manager -

Processor which performs all disk I/O, memory allocation, and memory deallocation for the CIG system.

MIMD -

See Multiple Instruction/Multiple Data Stream.

Mip Map -

An efficient memory arrangement of increasingly lower-resolution color maps. Mip is an acronym for the Latin "Multum In Parvo" - "Many things in a small space".

Mips -

Abbreviation for million instructions per second.

Model -

Generally used to refer to models of arbitrary, three-dimensional objects, such as buildings and vehicles.

Model Object Processor -

Processor which converts data defining a three-dimensional model in an arbitrary object space into a list of triangles in viewpoint space.

Model Space -

A 3-space in which a model or part is defined.

Model Tree -

Hierarchical database used to define a model in terms of its parts.

Model Tree Processor -

Processor which traverses the model tree and generates a list of parts to be processed.

MOP -

See Model Object Processor.

MOS -

Abbreviation for Metal-Oxide Semiconductor. A general technology for fabricating integrated circuits that uses three distinct layers to form transistors and interconnections.

MTP -

See Model Tree Processor.

Multiple Instruction/Multiple Data Stream -

A class of advanced computer architecture which is characterized by collections of connected processors executing concurrent processes.

Multiprocessor -

A computer with multiple arithmetic and logic units for simultaneous use.

N-buffering -

An extension of double-buffering to N buffers.

Nap-of-the-Earth -

Very low-level flight characterized by rapidly changing, highly-detailed imagery.

Need List -

In the scheduler, the list of nodes which is needed to process a frame.

NMOS -

A silicon-gate n-channel MOS technology that uses currents made up of negative charges and improves component performance.

Node -

A branching point in a tree structure. Also refers to a short descriptor of data residing at that node.

- Object Processors -**
Collective name for the Model Object and Terrain Object Processors.
- Orthographic Projection -**
Projection along a set of parallel lines.
- PDL -**
See Program Design Language.
- Pipelining -**
Refers to overlapping of execution cycles within a processor and is used to speed up a computer by performing operations concurrently.
- Pixel -**
Abbreviation for Picture Element. A pixel is the smallest addressable element of a video screen.
- Poking-through Effect -**
A tiling artifact whereby one polygon appears to poke through another.
- Polygon Coloring -**
Refers to assigning a single base color to an entire polygon.
- Polygon Table -**
In a scan-line hidden-surface algorithm scheme, a table that contains shading information, plane equation coefficients, and an in/out boolean flag for each polygon in the field of view.
- Pretiler -**
A processor which projects, clips, and computes depth and color gradients for a polygon.
- Program Design Language -**
A very high-level language used to design programs.
- Pseudo Pixel Projection -**
Approximate projection of a screen pixel onto a color map.
- PT -**
See Polygon Table.
- Quadric Surfaces -**
A class of curved surfaces which includes ellipsoids, paraboloids, and hyperboloids.
- Quadtree -**
A tree structure that contains four children per parent.
- RAM -**
Abbreviation for Random-Access Memory. RAM provides immediate access to any storage location point in the memory.
- Raster -**
A horizontal line of pixels on a display screen.

- Raw Gaming Area -**
Raw data required to construct a terrain model of some area.
- RAWGA -**
See Raw Gaming Area.
- Ray tracing -**
An exacting technique of generating an image on a display screen by tracing back through each pixel the light rays which ultimately reach the eye.
- Real-time -**
Pertaining to the performance of a computation during the actual time that the related physical process transpires in order that results of the computation can be used in guiding the physical process. In CIG applications, this is often used to specify a frame rate of 30 frames per second.
- Refresh Memory -**
A frame buffer containing the current displayed image used to generate the analog video signal.
- Render -**
To generate an image of a geometric entity from a mathematical model.
- Reprojection -**
The process of projecting a point from viewpoint space to screen space and back to viewpoint space again. Addresses various tiling problems.
- R,G,B -**
Red, green, blue.
- Root -**
See Tree Root.
- Scheduler -**
A processor which traverses the database tree to select the nodes within the field of view and at the correct level of detail.
- Scrambler -**
Part of the distribution tree which remaps memory addresses to evenly distribute data in the depth/frame buffer memory.
- Scan Conversion -**
The process of determining pixel values based on the transformed, clipped polygonal primitives.
- Scan-line -**
A method of processing that examines each pixel of a raster line sequentially.
- Screen Space -**
The two-dimensional viewing plane which defines an i,j coordinate system for the display screen.
- Separation Planes -**
Geometric planes used to separate objects or clusters for prioritization.

- Siggraph -**
Special Interest Group on Graphics.
- Stamp -**
See Color Map.
- Sun Angle -**
The angle that defines the direction of the sun.
- Surface Normals -**
The vector perpendicular to the surface.
- TED -**
See Digital Terrain Elevation Data.
- Terrain Elevation Data -**
See Digital Terrain Elevation Data.
- Terrain Object Processor -**
A processor that renders terrain by generating triangles from terrain data.
- Texture Map Space -**
The two-dimensional coordinate system used to address a color map.
- Texture Mapping -**
The projection of color maps onto geometric models.
- 3-Space -**
Three-dimensional space.
- Tiler -**
A processor that receives data describing an object, determines which screen pixels lie within that object, and calculates the color and depth of the internal pixels.
- Time Complexity -**
An algorithm is said to have time complexity $O(f(n))$ if the number of steps it needs to process data of "size" n is $cf(n)$, where $f(n)$ is some function of n and c is a constant.
- TIN -**
See Triangulated Irregular Network.
- TOP -**
See Terrain Object Processor.
- Transformation -**
A combination of rotation, translation, and scaling operations.
- Tree Root -**
The highest level of a hierarchical structure.
- Tree Structure -**
Hierarchical data structure.

Triangulated Irregular Network -

An irregular grid used to model irregular terrain surfaces.

TTL -

Abbreviation for Transistor-Transistor Logic. A technology for fabricating integrated circuits, the basic component of which is formed by interconnecting two bipolar transistors.

TTL Compatible -

Designates those integrated circuits where main logic does not utilize TTL technology, but which may interface with integrated circuits that do.

2-Space -

Two-dimensional space.

Universal Transverse Mercator -

Military mapping grid system for the earth.

UTM -

See Universal Transverse Mercator.

U,V,W Space -

See Viewpoint Space.

Vertex Coloring -

Coloring a polygon based on the colors assigned to each vertex.

Vertex Normals -

Weighted average of the face normals surrounding a polygon vertex. Approximates the normal to the surface at the vertex if the surface were curved.

VHSIC -

Abbreviation for Very High Speed Integrated Circuit. VHSIC is a military research and development program to develop a family of integrated circuits which operate at a speed faster than is currently available.

Viewpoint Space -

A 3-space defined relative to the viewpoint.

Viewpoint Vector -

A vector extending from a surface point to the viewpoint.

VLSI -

Abbreviation for Very Large Scale Integration. A term describing integrated circuits with a large number of transistors, usually in excess of 10,000.

Want List -

In the scheduler, a list of nodes which are desirable but not needed.

Whetstone -

A benchmark for floating-point performance of computing systems.

World Space -
A 3-space fixed relative to the world.

XGEN -
Old developmental image generation software package.

XYZ Space -
See World Space.

Yon Plane -
The plane which defines the field-of-view boundary farthest from the viewer.

Z -
The variable which defines the depth, or distance from the viewpoint, of an object.

Z-Buffer -
Buffer containing the distance from the viewer to the nearest surface, measured along a vector normal to the screen, for each pixel. See also Depth Buffer.

Z-Buffer Algorithm -
A hidden-surface algorithm which utilizes the z-buffer.

1.0 INTRODUCTION

1.1 RESEARCH OBJECTIVES

This report describes the results of a two-year program to investigate unique algorithms and hardware architectures for high-speed, high-complexity computer image generation (CIG).

Computer image generation is a subset of computer graphics which deals with creating perspectively correct views of an abstracted three-dimensional environment at high rates of speed. The ultimate goal of CIG is to produce photographic-quality pictures at a maximum rate of sixty pictures per second. The extreme computational load that results from producing high-complexity scenes in real or near-real time makes CIG the most demanding discipline in computer graphics today.

The program, sponsored by the Defense Advanced Research Projects Agency (DARPA), was the second phase of a three-phase effort. The first phase studied the feasibility of a CIG system architecture based on the z-buffer hidden-surface algorithm. The second phase implemented, refined, and tested this architecture as a pipelined, multi-instruction/multi-datastream system. In the third phase, the results of this research will be used to implement a high-complexity CIG system in a military setting.

DARPA sponsored this research as a result of their interest in developing innovative CIG systems for a variety of military applications. For optimal performance in a military setting, these CIG systems should be designed:

- o to achieve high throughput through multiprocessor architecture;
- o to be compatible with LSI/VLSI requirements to decrease cost, size, and power consumption, improve reliability, and exploit state-of-the-art technological advances;
- o to be durable in severe physical environments;
- o to be scalable across a range of display resolutions;
- o to reduce database construction and storage constraints;
- o to allow the database to contain arbitrary data types;

- o to permit rapid database assembly; and
- o to permit processing of non-visual object attributes for simulating images from a variety of sensor types.

The objective of the second phase was to test CIG algorithms and hardware architectures through the development of an architecture research system (ARS). This system was intended to be both an implementation of the first-phase concept of a CIG system, and a research tool that enabled different architectural concepts to be implemented and analyzed. The application chosen for the ARS was visual flight simulation with an emphasis on nap-of-the-earth flight. This application was viewed as providing the greatest opportunity to tax the design and implementation in terms of performance and scene complexity.

1.2 OVERVIEW OF THE ARS

The design strategy called for the ARS to consist of two major components. The first component is special-purpose hardware designed as a pipelined, multiprocessor system that implemented the key tasks of the z-buffer hidden-surface algorithm. The second component consists of CIG software on general-purpose computing equipment that performs the remaining tasks of the z-buffer algorithm and drives the special-purpose hardware.

The z-buffer hidden-surface algorithm is the key concept that forms the foundation of the ARS and dictates the design of the hardware and software. This algorithm performs the removal of hidden surfaces by dynamically comparing pixel depths and retaining color information for those pixels that are nearest the viewpoint.

The special-purpose hardware implements the z-buffer hidden-surface algorithm in a multiprocessor architecture. The actual execution of the algorithm occurs at the lowest level of the hardware in the depth/frame buffer memory. These eight memory modules receive pixel information that is generated by four tilters. The tilters receive triangle information from the CIG software and render these triangles into pixels. Connecting the tilters and the depth/frame buffer memory is the distribution tree, which is a multistage network that allows any tiler to send pixel information to any memory module. The antialiaser is the last section of

hardware in the processing path. It is designed to decrease undesirable visual effects caused by finite pixel resolution by oversampling a scene and then computing the final image by filtering the oversampled data.

The CIG software was intended to provide image generation capabilities as a front end to the ARS special-purpose hardware. CIG processing begins with terrain, model, and color information provided by a three-dimensional digital database organized in a tree structure. Data is communicated between the CIG software and the special-purpose hardware through the hardware-software interface (HSI). The HSI receives information from the pretilers. The pretilers compute screen-space information required by the hardware tilers from triangle data in viewpoint space provided by the object processors. These object processors receive raw terrain and model data along with viewpoint and sun position information, and generate triangle data that defines a three-dimensional, shaded, perspective-correct image. The raw data is provided to the object processors by the memory manager. The main task of the memory manager is to retrieve data requested by either the scheduler or model tree processor (MTP) and pass it along to the correct object processor.

The scheduler and the MTP are the highest level of the CIG software and work in tandem. The scheduler traverses the database node tree, selecting terrain and model nodes that are within the field of view and are at the correct level of detail. Model nodes that are chosen by the scheduler are passed to the MTP. The MTP traverses the model tree associated with each selected model node and selects additional nodes that increase the level-of-detail (LOD) complexity of the model. Model nodes from the MTP and terrain nodes from the scheduler are then fed into the memory manager.

1.3 STRUCTURE OF THE REPORT

This report details the process by which the design and implementation of the ARS were accomplished. The report begins with a discussion of the z-buffer hidden-surface algorithm. The next section describes the multiprocessor architecture that forms the framework of the ARS, and gives an overview of the design of the system, special-purpose hardware, and CIG software. The following two sections are devoted to a detailed technical discussion of the hardware and software design

and implementation. The hardware and software sections are presented using a bottom-up approach. This means that the discussion begins at the lowest level of the system where data exits the processing path, and progresses up the data path to the highest level where data is first generated. Section six is devoted to the results of the graphics algorithm research that was conducted in the areas of fractals, curved surfaces, transparencies, texture mapping, and level-of-detail control. The last two sections deal with system performance evaluations and the conclusions drawn from participation in this program.

2.0 THE Z-BUFFER ALGORITHM

The z-buffer algorithm forms the basis of Boeing's CIG system architecture. This chapter describes the algorithm and the reasons it was selected over other hidden-surface algorithms.

2.1 FUNDAMENTAL COMPLEXITY OF HIDDEN-SURFACE ELIMINATION

One of the most challenging problems in computer graphics is determining whether one object is obscured by another. Research over the past twenty years has yielded a number of algorithms which address this hidden-surface (or hidden-line) problem (Bouk69, Bouk70, Gene68, Gene71, Newe72, Romn70, Schu69, Warn69, Watk70, Wild71, Wyli67). Nonetheless, solving the hidden-surface problem remains the most difficult task performed by present day real-time CIG systems.

The fundamental complexity of the problem can be easily estimated from an analysis of the two basic classes of hidden-surface algorithms. One class of algorithms, known as object-space algorithms (Suth74b), attempts to prioritize surfaces before projection and scan-conversion, based on their distance from the viewer. To prioritize n surfaces, each surface may be compared to the $n-1$ remaining surfaces, yielding on the order of n^2 comparisons. For relatively complex scenes, such as $n=10,000$, approximately $n^2=10^8$ comparisons would be necessary.

The image-space algorithms (Suth74b), the other class of hidden-surface algorithms, determine which surface is visible at each resolution point of the display device. A display device with N^2 resolution points will therefore take nN^2 comparisons to render n surfaces. For $n=10,000$ and $N=500$, this results in 2.5×10^9 comparisons.

For real-time applications, the hidden-surface algorithm needs to be executed approximately thirty times per second. Therefore, approximately 3×10^9 object-space comparisons per second or 7.5×10^{10} image-space comparisons per second need to be performed for a real-time display of a complex scene. Regardless of the details of the hidden-surface algorithm, such execution rates are beyond the capacity of present-day general-purpose computers. Even when implemented in

special-purpose hardware, the hidden-surface algorithm consumes the majority of the computational power of real-time CIG systems.

2.2 HIDDEN-SURFACE ALGORITHMS

Four basic approaches have been taken to solve the hidden-surface problem. Each approach has resulted in a class of hidden-surface algorithms that share a similar framework, but differ in implementation details. An overview of each approach and its best-known algorithms is presented here.

The list-priority approach basically prioritizes surfaces according to some chosen scheme, and then performs projection and scan-conversion, allowing hidden surfaces to be overwritten. The depth-sort algorithm (Newell72) developed by Newell, Newell, and Sancha uses this approach. All polygons in the field of view are sorted according to their largest z-coordinate (see Figure 1). Then a series of complex tests are applied to resolve any priority ambiguities that result when the polygons' z-extents overlap. Next each polygon is converted into screen pixel information and placed into the image buffer in descending order according to each polygon's largest z-coordinate. The basic scheme behind this algorithm is to place polygons in the image buffer in order of decreasing distance. In this way, the nearest polygons are converted last, obscuring polygons that are further away by writing over them in the image buffer. The major disadvantage of this algorithm is that its order of complexity is fairly high due to the sorting required and the tests needed to resolve priority ambiguities. Also, the algorithm needlessly converts polygons that are obscured.

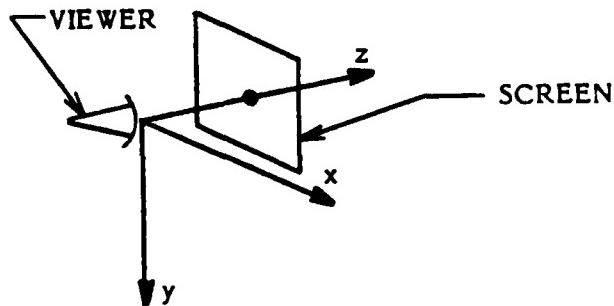


Figure 1. Definition of Z-Coordinate (Depth)

A highly ingenious algorithm that uses the list-priority approach was developed by Schumacker and his associates (Gene68, Gene71, Schu69, Wild71). This algorithm reduces the computational load on the CIG system by precomputing priority lists based on the topological properties of the environment. These priority lists form the framework of the database. Two observations are fundamental to this process. The first observation is that natural environments are characterized by clusters of faces sharing a common priority with respect to other clusters. For example, the clusters 1, 2, 3 in Figure 2 are separated by planes α , β . The cluster priority is based solely on the location of the viewpoint relative to the separating planes, and may be expressed in terms of a tree structure.

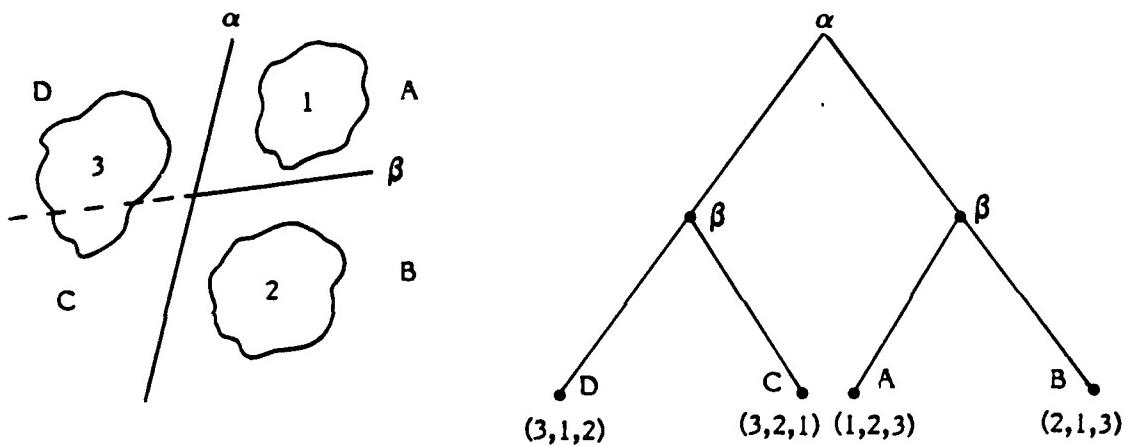


Figure 2. Cluster Priority

The second observation is that the priority of a face within a cluster is independent of viewpoint. Figure 3 illustrates this concept. Experimentation will verify the fact that after eliminating back faces, the precomputed face priorities correctly prioritize the remaining faces. The face's priority in the environment can therefore be represented as a decimal number $c.f$, where the integral part, c , is the cluster priority and the fractional part, f , is the face priority.

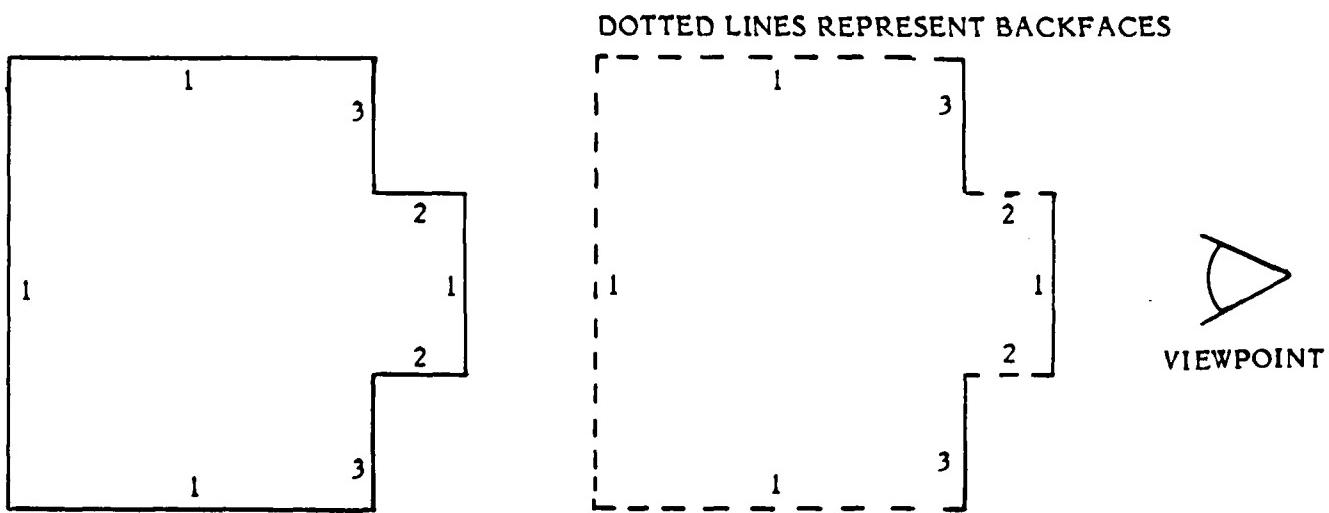


Figure 3. Face Priority

This algorithm renders a scene by finding the faces that cover segments of the scan-line and choosing the highest-priority face in each segment. The chief disadvantage of this algorithm is that construction of databases is very tedious. For example, the placement of separating planes is very time-consuming, and the database is very difficult to update.

The scan-line approach (Romn70, Wyli67, Bouk69, Bouk70, Watk70) performs a sort on face edges and processes this edge information as it traverses a scan-line. Scan-line algorithms construct and operate on three kinds of tables - an edge table (ET) for all non-horizontal polygon edges sorted by y-coordinate and slope; a polygon table (PT) which contains shading information, plane equation coefficients, and an in/out boolean flag for each polygon; and an active edge table (AET) containing all edges that intersect the current scan-line. Basically, the algorithms traverse a scan-line, processing the edges in the AET from left to right. Every time a new edge is encountered, the PT is queried to determine which polygon the scan is in, and what shading information is to be applied to the image buffer span. If the scan is determined to be in more than one polygon, the plane coefficients in the PT are evaluated to determine which polygon has the smallest z-value. The shading information for this polygon is then applied for that scan section. When a scan-line is completed, the AET is updated and ordered on x, and the next scan-line is processed. These algorithms have the advantage of converting only those polygons or sections of polygons that are visible, but their disadvantage lies in their

complexity. They sort in the y-direction to assemble the ET, the x-direction as each scan-line is processed and the AET is updated, and the z-direction to prioritize polygons. In addition, they continually evaluate the polygon plane equations to calculate z-values.

Another approach considered here is the area subdivision approach (Warn69, Weil77). Basically, subdivision algorithms examine an area in the image space. If the visibility of polygons in the area is easily determined, then the appropriate polygons are displayed. Otherwise, the area is subdivided into smaller areas and these smaller areas are scrutinized analogously. As the areas become smaller, they will overlap fewer polygons, and eventually a decision on visibility can be easily made. After each subdivision, individual tests must be made on all polygons that either intersected or were contained in the original area. Polygons that were disjoint from the original area or that surrounded the area need not be retested for they will retain their previous status. If subdivision has been carried down to the pixel level and no decision has been reached, then the depth of all polygons at the pixel will be computed. The nearest polygon will define the shading of the pixel. The major disadvantage of these algorithms is that the number of tests and calculations involved in area subdivision cause this method to be too slow for a real-time environment.

The approach chosen for implementation on the DARPA system is known as the z-buffer or depth-buffer approach (Fole82). It was first used by Edwin Catmull at the University of Utah to render bicubic Bezier patches that had been subdivided down to single pixels. Essentially, the z-buffer algorithm operates on two large two-dimensional buffers called the image buffer and the z-buffer respectively. The image buffer is used to store intensity values for each pixel on the output device. Similarly, the z-buffer stores z values or depths for each pixel. The first step in the process is to initialize the z-buffer to the largest representable z value and the image buffer to the background pixel intensity.

Next, each polygon in the field of view is converted into screen pixel information. During conversion, a depth and intensity is calculated for each pixel inside the polygon. The depth of each polygon pixel is compared against the depth currently stored in the z-buffer for that pixel. If the depth of the polygon pixel is less than the value stored in the z-buffer, it indicates that the current polygon is closer to

the viewer at that pixel than the polygon whose information was previously recorded in the z-buffer and image buffer. Therefore, the depth and intensity of the current polygon replaces the previously stored values for that pixel in the z-buffer and image buffer. If the current polygon is further away, as evidenced by a larger depth value, then that polygon pixel is considered hidden and its depth and intensity values are discarded. When all surfaces in the entire field of view have been processed, the z-buffer contains the smallest z values encountered for each pixel, and the image buffer contains the intensities associated with those z values.

The z-buffer algorithm has three major disadvantages. The first is the large amount of memory required for the z-buffer; this has become less of a concern as memory costs continue to decline and packaging becomes more dense (see Appendix A - subsection C). The second disadvantage is that hidden polygons are unnecessarily converted to pixels. The third and most significant disadvantage of current implementations of the z-buffer algorithm is that it discards all coherence information; for this reason, antialiasing becomes very difficult. This problem is discussed in greater detail in Sections 4.4 and 8.1.

Despite these concerns, the z-buffer algorithm has several very important advantages over other hidden-surface algorithms. These are discussed in the following section.

2.3 ADVANTAGES OF THE Z-BUFFER ALGORITHM

In the landmark paper, "A Characterization of Ten Hidden-Surface Algorithms," (Suth74b) Sutherland, et al. made an attempt to compare the cost of rendering scenes with differing complexities, using various hidden-surface algorithms. For this comparison, they hypothesized three environments: a simple one with 100 faces, an intermediate one with 2500 faces and a complex one with 60,000 faces. Their results are presented in Table 1.

Table 1. Relative Computational Costs of Different Hidden-Surface Algorithms (Suth74b)

Algorithm		100 faces	2500 faces	60,000 faces
List Priority:	Newell	140K	1.4M	71M
	Schumacker	4.2M	25M	170M
Scan-line:	Romney	770K	2.9M	14M
	Watkins	470K	3M	64M
Area Subdivision		1.5M	9M	43M
Z-Buffer		7.5M	7.5M	7.5M

From this comparison, it is clear that all hidden-surface algorithms except the z-buffer algorithm show enormous cost growth for highly complex scenes. In contrast the cost of the z-buffer algorithm remains constant, since the faces decrease in size as they increase in number. These performance characteristics therefore make it uniquely suited for rendering very complex scenes.

Another advantage of the z-buffer algorithm is that surfaces can be placed into the buffer in a random order. This allows a number of processors to process surfaces independently and place them into a single, shared z-buffer, providing the ideal environment for multiprocessing. A multiprocessor implementation of the z-buffer algorithm would increase throughput and enable complex scenes to be rendered more efficiently.

The z-buffer also places very few constraints on the types of objects that can be rendered. Any surface representation which can be converted to pixels may be used. By virtue of this flexibility, one can render quadric surfaces, b-spline patches, or particle systems as easily as polygons. In addition, traumatic geometric constructs such as cycles and penetrating surfaces present no difficulty (see Figure 4).

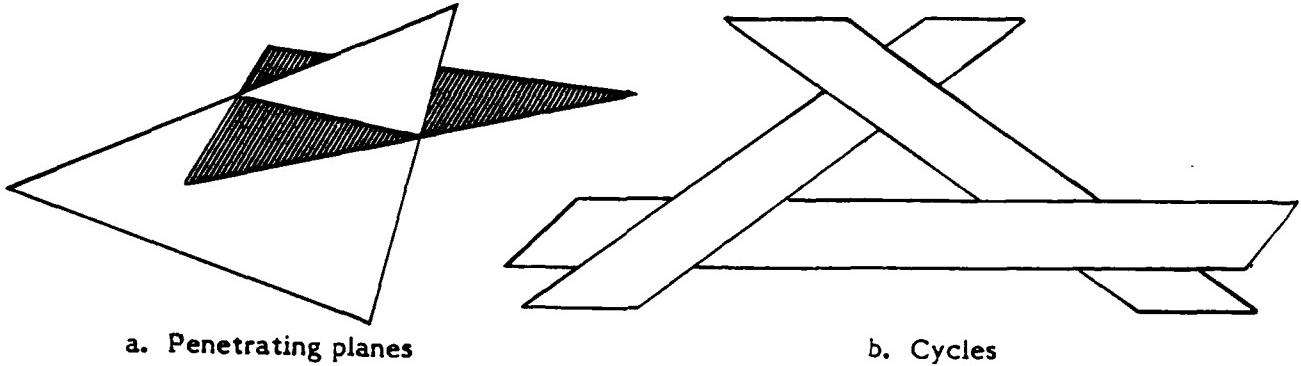


Figure 4. Traumatic Geometric Constructs

The final and most significant advantage of the z-buffer algorithm is that it places no constraints on the database structure. This permits the database to be structured specifically for the task at hand, and to be built rapidly and with little or no human intervention. The performance criteria might be rapid retrieval for flight simulation, rapid data entry for intelligence applications, or relational linkages for an information retrieval and presentation system. Any of these database structures can be utilized by the z-buffer algorithm.

Because of these advantages, a growing proportion of the computer graphics community is turning to the z-buffer for hidden-surface elimination. Its use in real-time systems, however, has been limited due to the high storage and throughput requirements.

The next section will discuss a system architecture which addresses these problems, and thereby captures the full benefits of the z-buffer algorithm.

3.0 SYSTEMS DESIGN

The architecture research system (ARS) is designed as a multiprocessor implementation of the z-buffer hidden-surface algorithm. Special-purpose hardware was constructed to perform the key tasks of this algorithm, and the remaining tasks were implemented in software on general-purpose computing equipment. This section begins with a detailed discussion of the multiprocessor architecture that forms the foundation of the ARS. Because the system was designed as a research tool rather than a real-time simulator, several compromises were necessary. These compromises and their influence on system design and performance are explored next. An overview of the system and a presentation of the key elements of the hardware and software designs conclude this section.

3.1 MULTIPROCESSOR ARCHITECTURE

A hardware implementation of the z-buffer algorithm can utilize the fact that polygons need not be prioritized to create a system with a high-throughput, multiprocessor architecture (ParkF80). The special-purpose hardware is designed as a parallel, pipelined machine of the multiple instruction/multiple data stream (MIMD) type. Pipelining at the module level is utilized to separate major functional tasks, allowing each task to run at maximum efficiency. High polygon and pixel throughput is achieved by allowing processing to proceed on any of the identical parallel processing paths. The scrambler and distribution tree distribute data from the four parallel tilers evenly to the eight parallel depth/frame buffer memories.

Pipelining within the system occurs at the module level as well as internal to the modules. To generate an image, thousands of polygons and millions of pixels are processed. The various processing steps are quite unique and require different types of hardware to run efficiently. Pipelining at the module level allows each distinct operation to be implemented in the hardware that will most effectively execute it. All pipeline stages are operating on a different piece of data. Therefore, no hardware is idle, and maximum system throughput is achieved. Pipelining internal to the modules achieves the same effect within the module.

Some stages operate at a constant rate; however, many operate at a rate which depends upon the exact data received. To allow smooth data flow, with minimal time spent waiting to send or receive, data queues are placed between the pipeline stages (see Figure 5). The queues are built with first-in/first-out (FIFO) memories that allow independent input and output, so that stages can operate asynchronously.

Maximum efficiency and parallel system operation are obtained with the scrambler and distribution tree. To handle the large number of polygons and pixels that are needed to generate a highly detailed image, many identical processing paths are utilized. The obvious method of dividing the load among parallel processing paths is to divide the display into equal areas and have each processing path handle an area. This method requires that the host decide along which path the data needs to be directed. Such a decision is non-trivial. It also requires that objects which cross area boundaries either be double-processed or clipped. Either of these

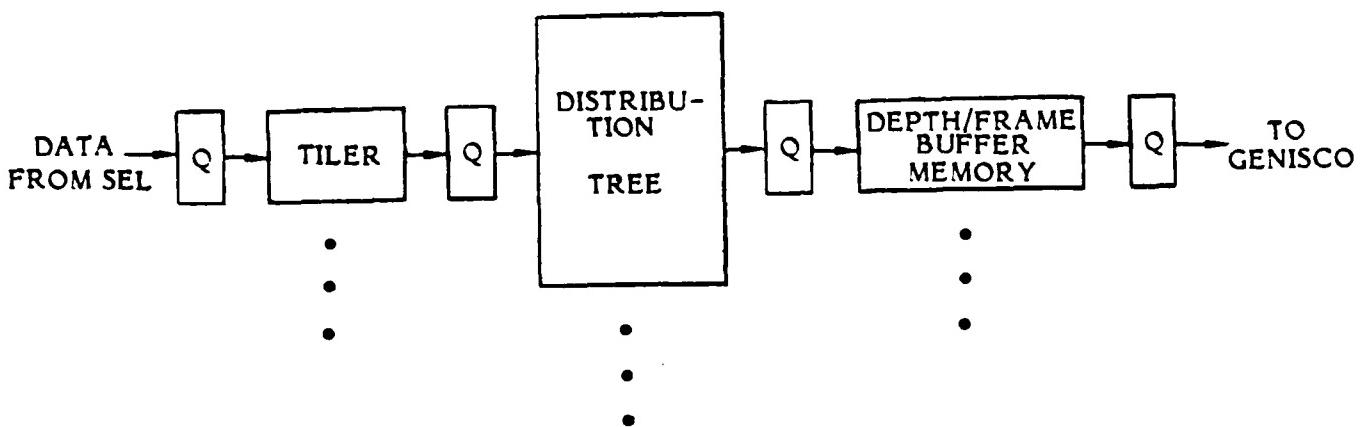


Figure 5. Queues Used to Smooth Data Flow

options leads to inefficiencies. For these reasons, it was decided to allow polygon data from any portion of the display to pass through any of the parallel processing paths. Pixels generated by the processors are routed to the correct depth/frame buffer memories by a distribution tree.

Scrambling of pixels is used to achieve a uniform distribution among the depth/frame buffer memories. If scrambling was not utilized, then memories would be assigned to areas on the screen, as in Figure 6, and two major problems would occur. First, the tiler would output pixels in a non-random fashion, causing the distribution tree to operate inefficiently. More importantly, because data is not evenly distributed over the image, some memories will be overburdened. For example, memory 4, which is along the horizon where the depth complexity is high, will be heavily accessed. Other memories, such as memory 0, will be lightly accessed, since it has only sky and low depth complexity. Scrambling breaks an image into small areas, as shown in Figure 7, to allow uniform pixel distribution. In

the ARS hardware, scrambling areas are only one pixel, with the pattern repeating in the i and j directions.

A major advantage of the parallel processing method used in the architecture research system is its fail-soft capability. If a processing path should fail, system throughput will be degraded, but the remaining parallel paths can process the data that the failed processor would otherwise handle.

The parallel, pipelined architecture developed for the architecture research system is capable of being upgraded for future, more complex systems. To handle real-time update rates, functions that are currently simulated in software, such as backface elimination, viewpoint transformation, and screen projection, will need to be implemented in hardware. These functions will be added as new pieces of the processing pipelines. Polygon throughput can be increased by enlarging the distribution tree and increasing the number of tiler paths. The parallel nature of the distribution tree also allows data generated from other sources to be entered into the depth/frame buffer memory. As such, systems can be easily configured to meet the specific needs of a variety of users.

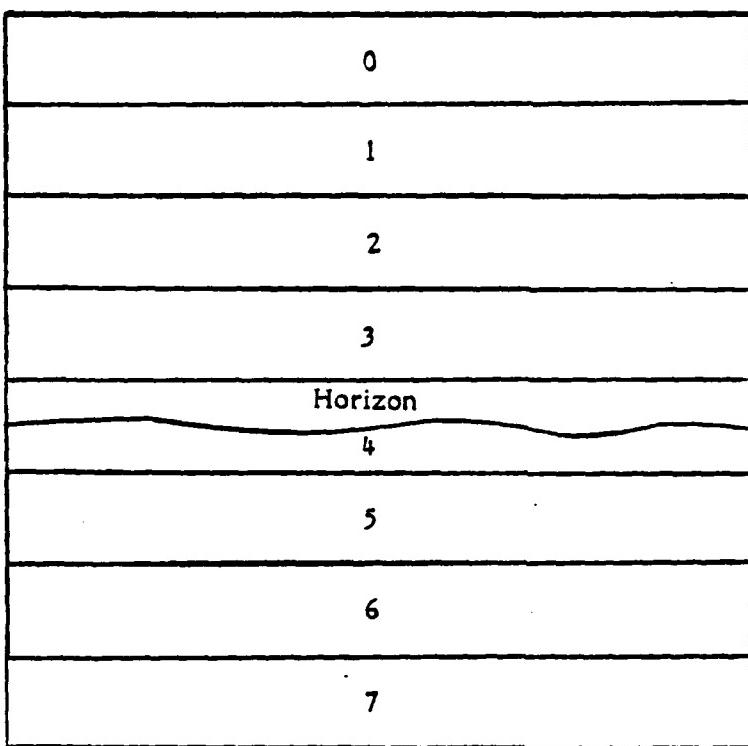


Figure 6. No-Scrambling Illustration.

0	3	6	1	4	7	2	5
1	4	7	2	5	0	3	6
2	5	0	3	6	1	4	7
3	6	1	4	7	2	5	0
4	7	2	5	0	3	6	1
5	0	3	6	1	4	7	2
6	1	4	7	2	5	0	3
7	2	5	0	3	6	1	4

Figure 7. Scrambling Illustration

3.2 COMPROMISES

The computer image generation system for this project was developed as a z-buffer architecture research tool, rather than a real-time simulator. To reach this goal in the most cost-effective manner, several compromises were necessary. Their purpose was to reduce the cost of hardware design and documentation. Reducing these costs enabled more algorithmic research to be performed.

In order to reduce the amount of hardware that would be built, it was decided to sacrifice several things needed for real-time operation. To do this, the entire hardware system works on only one color word at a time, so three complete passes are needed if a full 24-bit color definition is desired. Processing one color at a time eliminated the need for two additional color-accumulation paths in each tiler, reduced the width of the distribution tree by 16 bits, and allowed a 16-bit reduction in the memory word. All relevant statistics may still be obtained while running a single color through the system.

A Genisco image processor replaced double buffering of the memory and the use of analog circuitry to drive the video monitor. After a frame is calculated, it is digitally downloaded to the Genisco, which acts as a frame buffer and contains the refresh circuitry. The transfer takes approximately 400 milliseconds and is noticeable on the monitor.

The processing speed of this system is limited by the software throughput rate and data transfer rate between the host minicomputer and the special-purpose hardware. The software system was designed to be extremely flexible, allowing for implementation of new algorithms and ideas with a minimum amount of difficulty. To meet this goal, many "tricks" that would increase throughput were not utilized.

The hardware performs approximately half the tasks needed to generate an image with the remainder being done in the host. Before a real-time system is developed, a memory manager, object processors, and pretiler will need to be designed in hardware. Because the data formats utilized in the host SEL 32/87 and the hardware are not the same, data must be converted before it is transferred to hardware. This conversion takes a considerable amount of time, which limits the transfer of data.

Shortcuts were made in the method of shading triangles. Because of the perspective transformation, color shading over a triangle is not linear in screen space (see Section 4.2). The implementation of correct color shading would require additional hardware, and in most cases would not appear visibly different. It was therefore decided to assume color was linear in screen space for the design of the present system.

The system is designed and built to commercial standards and documented to research and development standards. All hardware parts are commercial quality, with no military specification requirements. This reduces the cost of the parts without degrading performance as a laboratory tool. The design was performed to worst-case commercial standards. This will be adequate for the temperature range in which the hardware is expected to be used. Configuration control was implemented in both the hardware and software designs. Hardware controls include an Interface Control Document, drawing package standards, and microcode standards. Software was documented in a Program Design Language before any code was written.

These compromises were made to reduce the overall cost of the research program. They did not, however, prevent the use of specialized test procedures to evaluate eventual real-time operation. Thus, all hardware was designed to operate at rates sufficient to sustain real-time performance, and specialized software was used to test the system's throughput under real-time conditions.

3.3 SYSTEM OVERVIEW

The DARPA computer image generation system consists of the specially designed architecture research system and additional standard peripheral equipment. A block diagram of the entire DARPA system is shown in Figure 8. The specially designed ARS is illustrated in more detail in Figure 9. The CIG software allows complex images to be generated and various graphics algorithms to be investigated. The data-recording equipment enables single images or sequences to be recorded for display and analysis. The special-purpose hardware was built to examine and prove the multiprocessor approach to z-buffer hardware design.

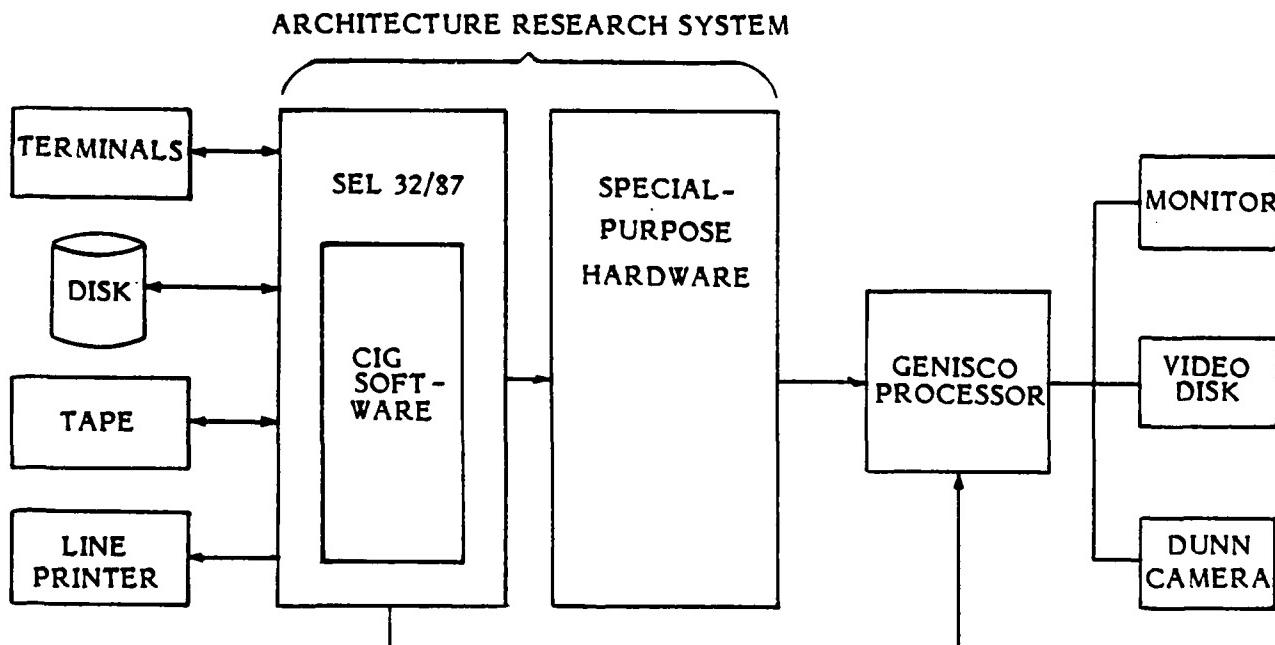


Figure 8. DARPA System

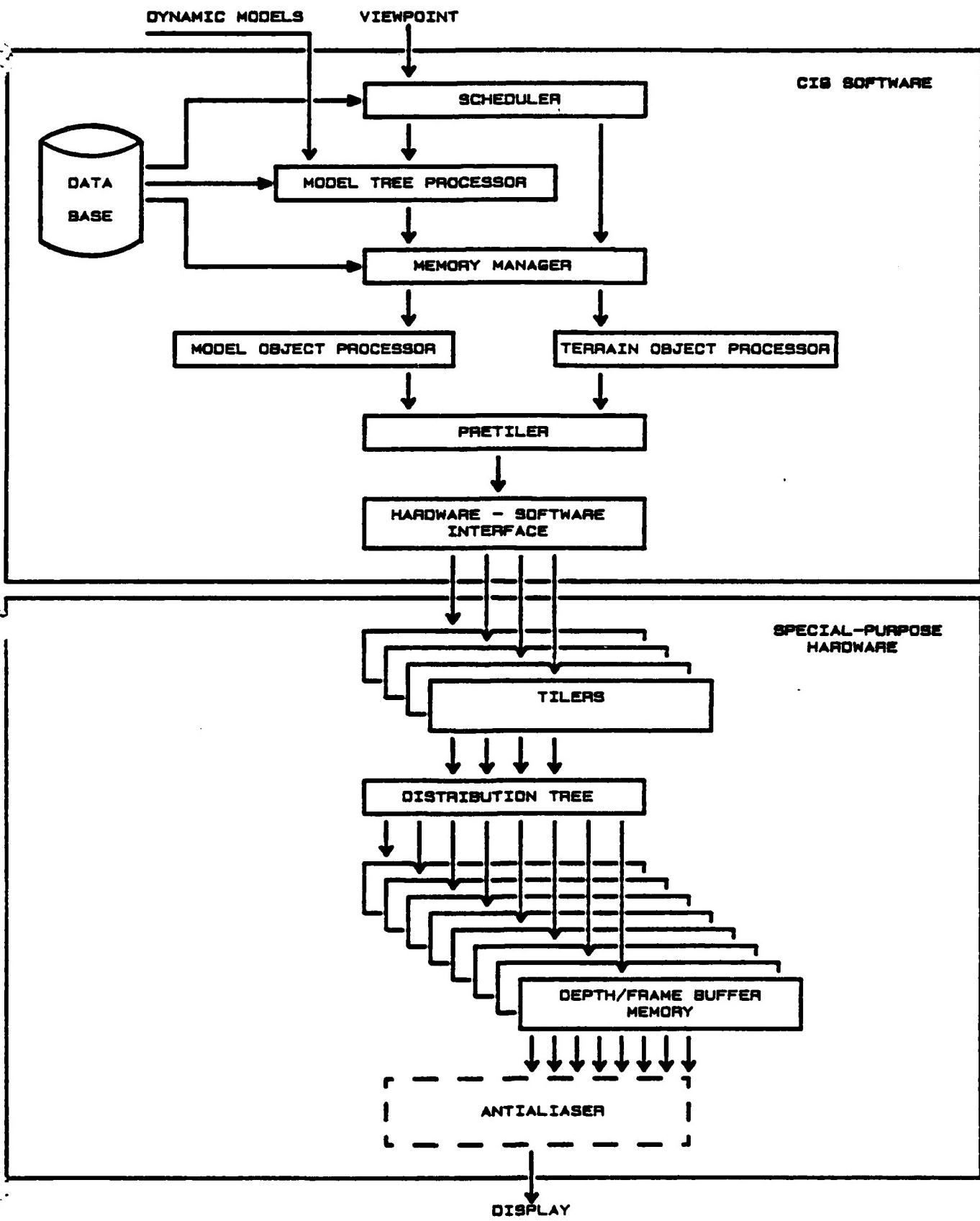


Figure 9. Architecture Research System (ARS)

3.3.1 DARPA SYSTEM CONFIGURATION

Several pieces of hardware were purchased and integrated into the DARPA system. The heart of the system is a Gould/SEL 32/87 minicomputer equipped with 4 Mbytes of memory, a tape drive, a 300 Mbyte disk, seven High Speed Data (HSD) interfaces, and eight user terminals. All system software is developed and run on the SEL. A Genisco display processor is interfaced to the SEL via an HSD interface. Images generated on the SEL or by the custom hardware can be stored in the Genisco refresh memory and displayed. Two pieces of hardware are utilized to record images. Hardcopies are obtained with a Dunn Instruments camera system which is equipped with an 8x10 Polaroid hardcopy unit and 35 mm single-frame and 16 mm movie cameras. An Echo Science video disk recorder with monitor was connected to the SEL via a custom interface and allows images to be recorded and played back in real time.

3.3.2 CIG SOFTWARE

CIG software can be divided into two categories - the software designed for basic image generation and the software implemented for graphics algorithm research. The basic system will allow the processing of triangulated data bases. DMA terrain elevation data is utilized to generate terrain profiles. Cultural features and models are generated by processing triangulated objects stored in the data base. The system allows triangles to be processed with face or Gouraud shading and variable sun angle. Basic system software includes database construction tools, scheduler, model processor, memory manager, terrain model processor, model object processor, pretiler, and hardware/software interface. Images can be created without the aid of the special-purpose hardware by utilizing software tiler and depth-buffer programs. Graphics research software was developed to investigate fractals, curved surfaces, transparencies, texture mapping, level-of-detail control, and antialiasing.

3.3.3 SPECIAL-PURPOSE HARDWARE

The special-purpose hardware in the ARS is a multiprocessor design, capable of processing shaded triangles (tiling) and performing hidden-surface elimination on complex images at high speed. The system contains four tilers that each interface

to the host SEL over an HSD, a four-to-eight scrambler and distribution tree, eight depth/frame buffer memories, a control/status board that controls system operation and gathers statistics, and an interface to the Genisco Graphics Processor. Processing that must occur before tiling (data base retrieval, backface elimination, viewpoint transformation, etc.) is performed by the CIG software in the SEL. The data is then formatted and shipped to the tilers.

3.4 HARDWARE OVERVIEW

The ARS special-purpose hardware is a multiprocessor implementation of the key elements of a z-buffer CIG system. It was built to gather and analyze statistics relevant to the multiprocessor architecture, and to prove the feasibility of the z-buffer design approach. The special-purpose hardware consists of four tilers, a four-tiler-to-eight-memory-module scrambler and distribution tree, and eight depth/frame buffer memories (see Figure 10). It also includes a display interface and a control/status module (not shown in the diagram). All hardware is wire wrapped on 14 by 17 inch Mupac circuit cards. Bipolar (TTL and TTL compatible) and NMOS digital integrated circuits were utilized in the design. Two card cages, one holding the eight tiler cards and the other holding the ten remaining cards, are mounted in one rack of the system. A second system rack contains two 5 volt/300 amp power supplies and a monitor. The system contains approximately 7500 integrated circuit chips, processes a maximum of 400,000 triangles per second, tiles and performs hidden-surface elimination on 40 million pixels per second, and draws approximately 280 amperes of power.

Hidden-surface elimination and frame data storage is accomplished in the depth/frame buffer memories. The memories receive a pixel defined by screen coordinates, color, and depth. The screen coordinates are used to address the memory. The basic cycle of the z-buffer hidden-surface elimination is comparing the new depth value with that stored in memory. The depth and color data is overwritten only if the new data is closer to the viewpoint than the old data. Since the frame buffer is not double-buffered, color values are read from memory by the Genisco interface card after the entire frame has been processed.

A tiler receives data describing a triangle, determines which screen pixels lie within the triangle, and calculates the color and depth of the internal pixels. Each tiler is made up of four pipelined stages. The first stage is an interface to the host SEL minicomputer over a 32-bit parallel HSD data bus. This interface passes eight data words describing the triangle in a worst-case time of 9.6 microseconds, or 1.2 microseconds per transfer. Data is placed in a FIFO queue as it is received to await processing. The second pipeline stage is called tiler setup. Here two microcoded arithmetic processors perform calculations common to the entire triangle, within approximately 8 microseconds. Data is then passed from tiler

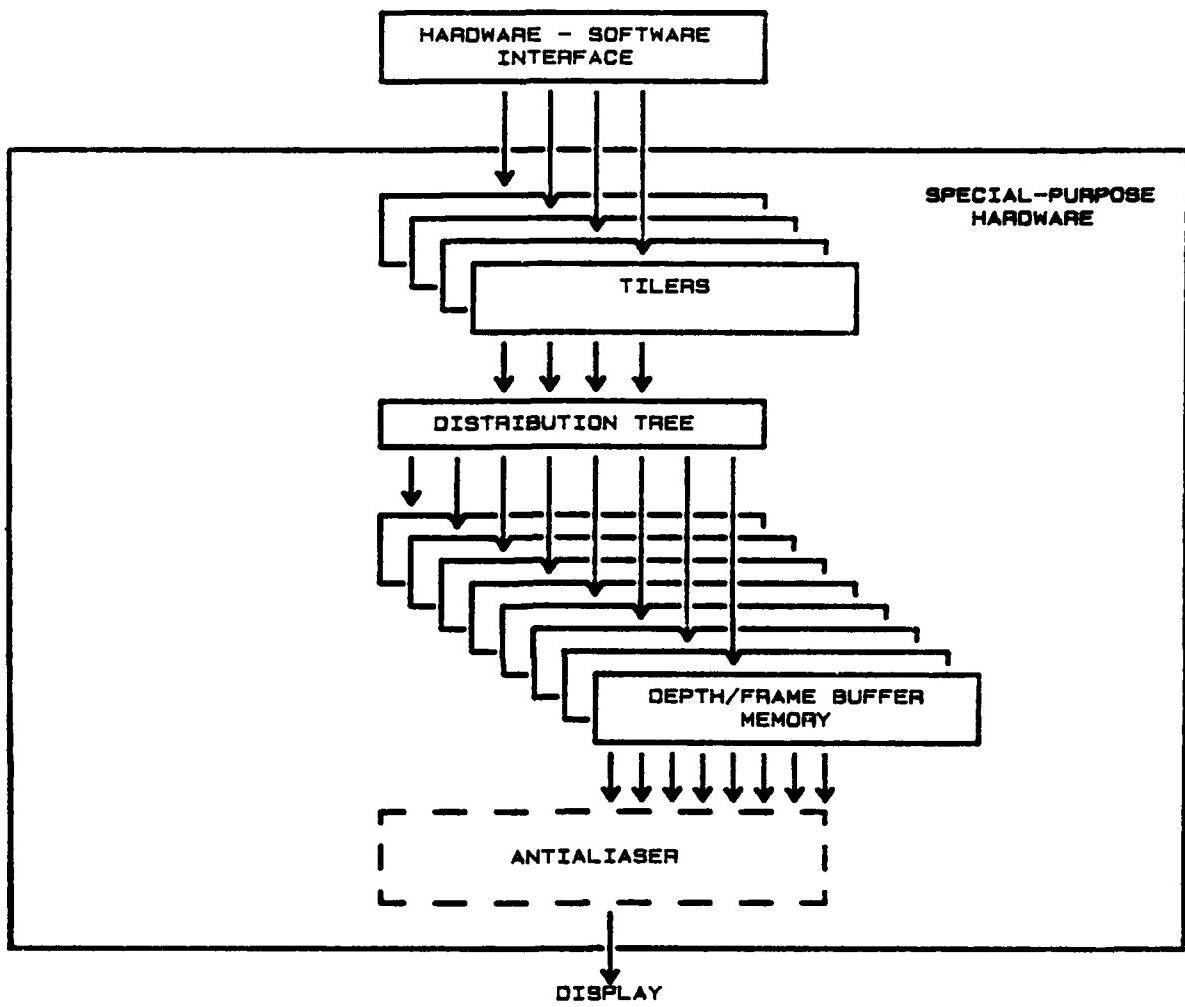


Figure 10. Special-Purpose Hardware

setup to the third stage, the tiling machine, where a triangle is actually "tiled". This involves testing pixels to determine if they are internal to the triangle. Incremental depth and color values are calculated at the same time. The last stage, tile accumulation, adds the incremental floating-point numbers together to obtain a final depth and color value for each pixel. Internal pixel data is then passed to a FIFO queue for output to the distribution tree. Pixel calculations are performed at a 10 MHz rate, which means pixels are tested every 100 nanoseconds. Because the tiling process is not 100% efficient, the actual pixel output rate will be less than 10 MHz.

The scrambler and distribution tree allow any tiler to pass data to any memory module, while maintaining a nearly uniform distribution of data to the memories. The distribution tree is a set of four 1x2 switches followed by two pipelines of four 2x2 switches. The routing bits, three bits that route a pixel to the correct memory module, are calculated in the 1x2 switches. Three modes are possible. The sequential mode involves choosing the three most significant bits of i, which divides the screen into eight equal horizontal bands. The interleave mode, obtained by choosing the three least significant bits of i, routes each raster line to a different memory (Modulo 8). Scrambling, theoretically the best method, involves combining the least significant bits of i and j. To smooth data flow, queues are placed in each 2x2 switch. Eight FIFO queues are placed at the end of the tree to allow asynchronous operation between the tree and the memory modules. The tree accepts data at 10 MHz and passes it between pipeline levels.

The remaining hardware tasks are performed by the Genisco interface and the control/status module. Color data is passed from the memory modules to the Genisco display processor under the control of the Genisco interface card. A hardware antialiaser was designed to fit on this card, but research into antialiasing methods revealed that a greater amount of oversampling is needed to obtain satisfactory results. Therefore, it was decided not to add this circuitry to the hardware. The control/status board interfaces to the SEL minicomputer over a bidirectional HSD interface. The SEL can control hardware operation by issuing commands, such as "clear system" and "frame complete", and can tell the board which statistics to gather. Special-purpose counters are utilized to gather statistics on the tilers, distribution tree, and memories. At the end of a frame, statistics are sent to the SEL for analysis. These statistics are used to determine optimal system performance and to assist in defining future systems.

3.5 SOFTWARE OVERVIEW

3.5.1 SYSTEM DESIGN

The DARPA CIG software is based on previous software efforts, including a Boeing computer graphics research system known as "XGEN". This system was developed to begin research into depth-buffer-based computer graphics. XGEN has been used on various projects such as missile basing schemes, a texture-mapping study for NASA, fractally-defined geometry, computer-generated terrain maps, and simulated radar imagery. The base of knowledge developed during the evolution of XGEN allowed the DARPA software to be designed in a top-down, modular fashion. In addition, many of the algorithms were reexamined and greatly refined as they were transferred to the DARPA software.

The goal of the DARPA software was three-fold. It drives the hardware, and as such emulates the front end of a real-time processing chain. It investigated how that processing chain should be modularized for eventual hardware implementation, and it allowed a flexible framework for further graphics research.

An overview of the DARPA CIG software system appears in Figure 11. It consists of (from back to front): the hardware/software interface, the pretiler, the model object processor and terrain object processor, the memory manager, the model tree processor, the scheduler, and the database. Not shown are a software emulation of the depth/frame buffer memory and the tiler. Also not shown are additional modules used for graphics research, such as the B-spline processor and texture mapper. The following sections will explain the functions of the basic processors shown in Figure 11.

3.5.2 HARDWARE-SOFTWARE INTERFACE (HSI)

The HSI provides communication between the pretiler software and the hardware. The HSI accepts triangle data from the pretiler, formats the data, and handles all transfers to the ARS hardware. Control signals required by the hardware are also handled by the HSI.

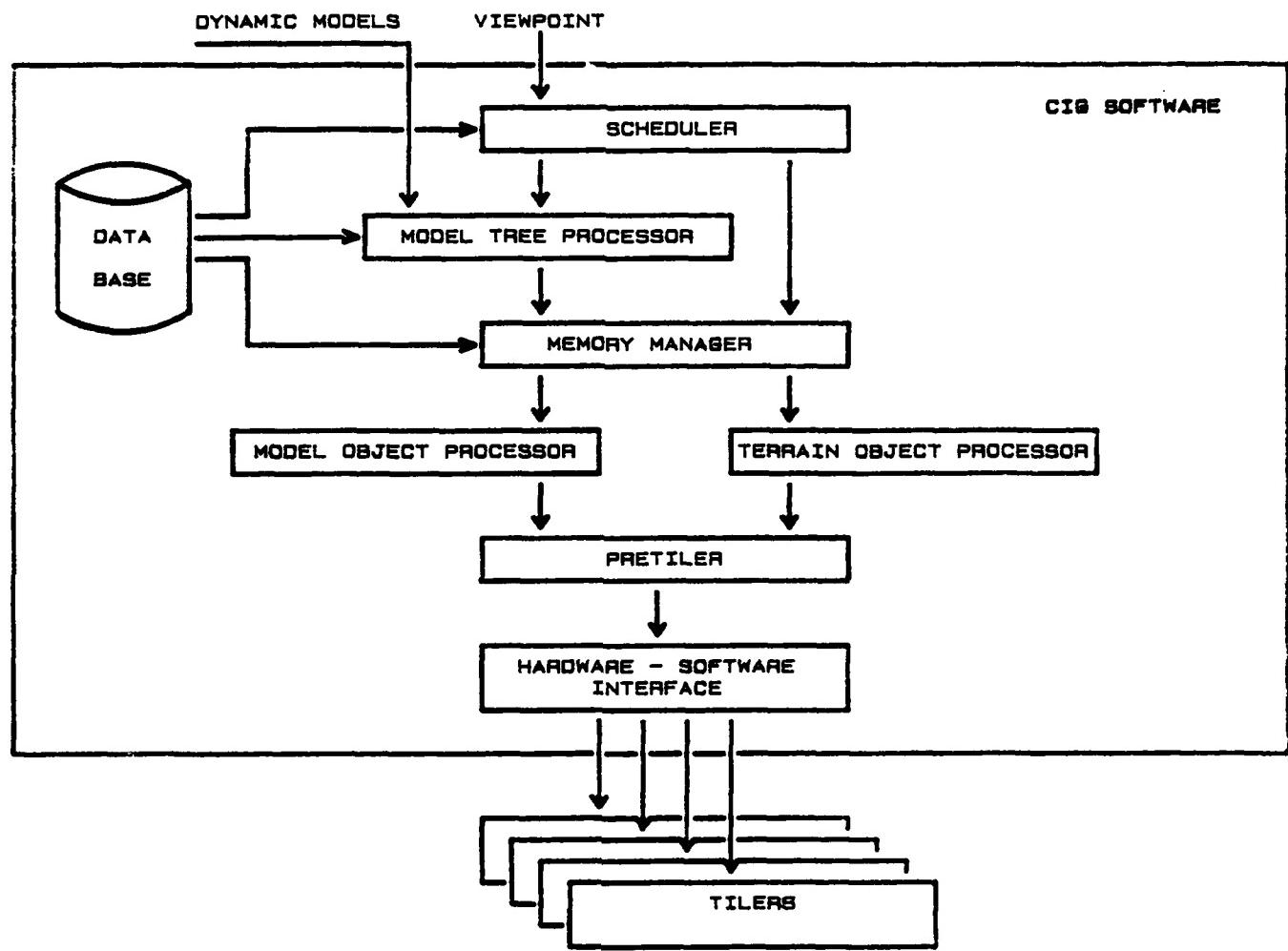


Figure 11. CIG Software

3.5.3 PRETILER

The pretiler accepts three vertices specifying color and location in 3-space. This processor transforms the data into lower level information used in the tiling processor. This lower level information consists of the screen-space coordinates of the triangle vertices, the (red, green, blue) color gradients of the triangle, and the depth gradients of the triangle. The pretiler is also responsible for clipping the triangle to the viewing volume.

3.5.4 OBJECT PROCESSORS

The model object processor and terrain object processor transform data from an object into a list of triangles which are passed to the pretiler. The model object processor is the most generic of the object processors and will process any 3-D geometric model. The terrain object processor takes advantage of the regular grid structure of the terrain data to improve the throughput of the system.

3.5.5 MEMORY MANAGER

The object memory manager is responsible for intelligent allocation and deallocation of data memory. The major design goal was preserving frame-to-frame coherence of data.

3.5.6 MODEL TREE PROCESSOR

The function of the model tree processor is to traverse a model tree and generate a list of model tree nodes which point to the model headers. The major difference between model tree traversal and database tree traversal is the concatenation of transformation matrices in the model tree. These transformation matrices allow arbitrary positioning and motion of model parts.

3.5.7 SCHEDULER

The scheduler traverses the database tree, based upon the field-of-view and level-of-detail tests. The scheduler generates lists of nodes to be processed which are passed to the memory manager or model tree processor. These lists are sorted by

distance from the viewer to guarantee that the nearest objects are processed in case of a time-out condition.

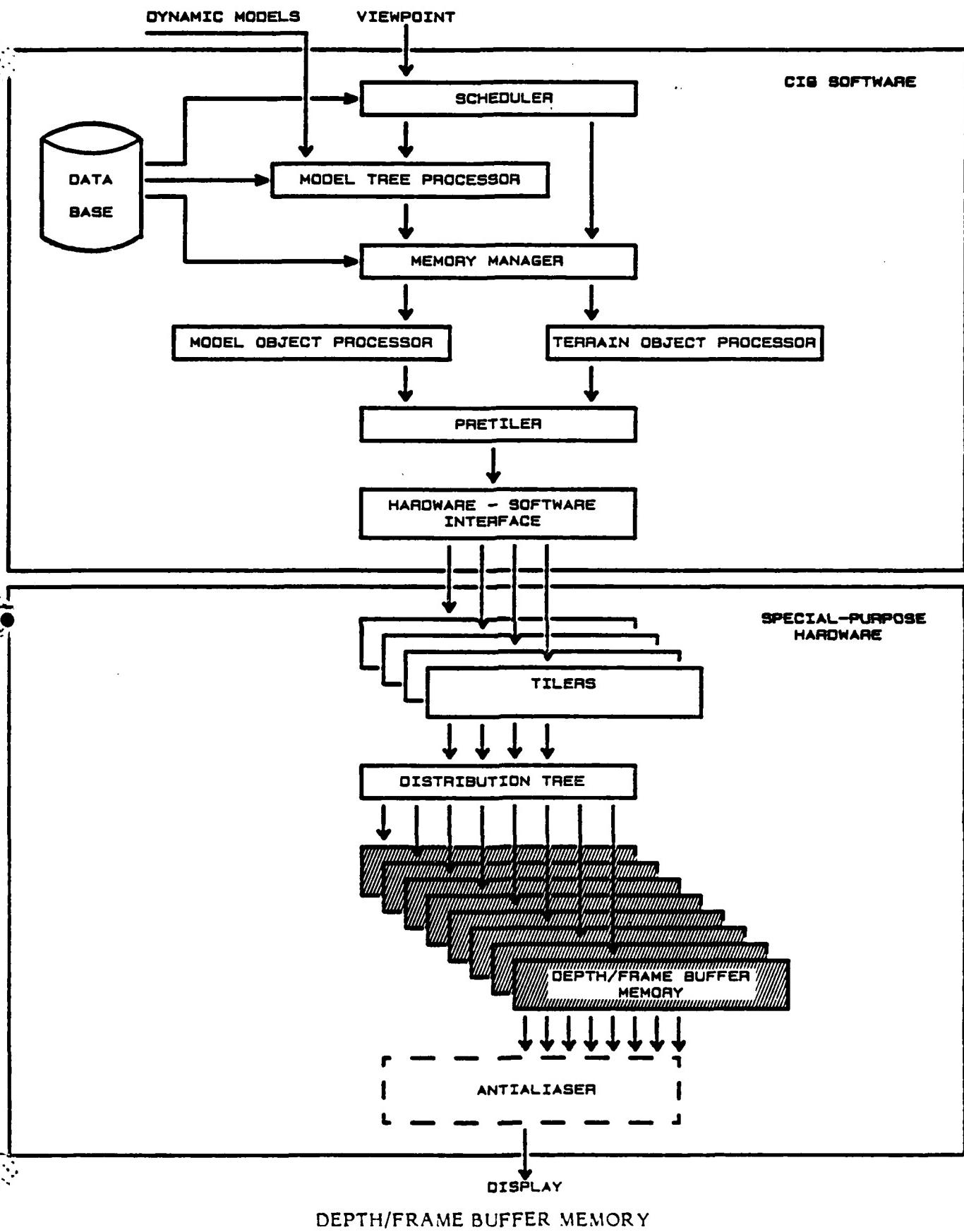
3.5.8 DATABASE

The main design goal for the gaming area database was generality. The database must not be restricted to specific types of data (for flexibility), and the structure should allow modeling of both large and small areas. Also, the data should be ordered to allow large numbers of objects or areas to be discarded if not in the field of view (FOV). This allows for a minimum of FOV checking and will reduce the transport delay in the system.

These design criteria were met by using a hierarchical glossary tree structure to define the gaming area. The tree contains field-of-view and level-of-detail (LOD) information and is ordered according to LOD. The root of the tree contains the least amount of detail and the leaves contain the most detail. This fulfills the ability to model both large and small gaming areas. Expanding the scope of the database merely involves increasing the size of the tree.

4.0 HARDWARE DESIGN

The following section examines the algorithm development, detailed design, and implementation of the special-purpose hardware. This custom hardware is at the heart of any CIG system. It is the system component that quickly and efficiently creates displayable images from the building blocks of polygonal data. This section begins with a short discussion on the design and implementation of depth/frame buffer memory which combines image storage with an implementation of the z-buffer hidden-surface algorithm. The tilers and the distribution tree that connects the tilers to depth/frame buffer memory are presented next. A short section on the antialiaser concludes the hardware design section.



4.1 DEPTH/FRAME BUFFER MEMORY

4.1.1 BASIC CONCEPTS

Depth/frame buffer memory is custom hardware that performs hidden-surface elimination by an implementation of the z-buffer algorithm (see Section 2.0) and stores color and depth information for each pixel in the display image. Depth/frame buffer memory receives pixel address, color, and depth information from the tilers via the distribution tree. To determine which pixels are visible, it uses a fast floating-point compare network to compare the depth value of the incoming pixel with the depth information already stored in memory for that pixel. If the new depth is closer than the stored depth, then the new pixel is considered to obscure the old pixel. The new color and depth information then replaces the old information in memory. If the old depth is closer, then the new information is discarded. Depth/frame buffer memory therefore always contains the color and depth information for the nearest pixel processed for each pixel location. When processing by the tilers is complete, the stored pixel information is output via the antialiaser to the Genisco display processor.

4.1.2 HARDWARE DESIGN OBJECTIVES

The key concern in the development of depth/frame buffer memory was speed. With this objective in mind, two implementation strategies were pursued - asynchronous operation and variable cycle lengths.

The depth/frame buffer memory controllers are asynchronous, as well as their interfaces to the distribution tree and antialiaser. Decisions are acted upon as soon as they are made, instead of waiting for a synchronous clock pulse to allow the action to occur. This allows memory throughput to be optimized.

Variable length cycles save time by taking advantage of some shortcuts that are inherent to the z-buffer algorithm. The longest process in the algorithm occurs when the new pixel is compared to the stored pixel and is found to have a lesser depth. Then, a write must occur to replace the old information. But if the new pixel is compared and found to be further away, the cycle can terminate without a write. A more trivial case occurs when a pixel location is being written to for the

first time. No comparison is needed so the cycle consists of an automatic write. To fully optimize the implementation of the z-buffer algorithm, seven cycles were identified. These cycles are single, independent processes that are chosen for initiation by the memory controller depending on flag values and the results of floating-point comparisons. These cycles are more fully described in the following hardware design overview.

4.1.3 HARDWARE DESIGN OVERVIEW

The depth/frame buffer memory is composed of eight identical memory modules that operate independently. Each module has a storage capacity of 32K 33-bit words. A word in memory consists of 24 bits of depth information, 8 bits of color information, and a 1-bit initialization flag. The total system memory capacity is 256K words which is the size needed to produce a 512x512 pixel image.

High-speed TTL logic was used for the control circuitry. The RAMs that were employed were fast, static N-channel MOS RAMs organized 16Kx1. This organization allows for a worst-case access speed of 45 nanoseconds.

The z-buffer algorithm is implemented as seven different cycles, one read cycle and six variable-length write cycles. The write cycles are controlled by the results of the floating-point comparison network and a series of flags. The longest cycle is less than 200 nanoseconds, matching the rate at which data is optimally sorted to the module.

The background (BG) cycle was designed for situations in which memory needs to be unconditionally overwritten with incoming data values. These situations occur when memory must be filled with a background such as sky or when information such as grid lines or symbols are overlaid onto the image. The BG cycle is paired with a background flag. This flag is controlled by the host computer and is sent, along with the specified memory location, to the memory controller via the tiler and distribution tree. When the memory controller receives a true value for the background flag, it initiates a background cycle. This cycle performs an automatic overwrite to memory. The contents of the specified memory location are replaced by the incoming data.

The initialization (INIT) cycle is paired with an initialization flag. This flag is carried as 1 bit in the memory location word. It indicates whether data has been previously written to that memory location. When the memory controller addresses a pixel location, it checks this flag. If the flag is set to false, that pixel location is recognized as empty. Instead of initiating a long compare cycle, the controller will initiate an INIT cycle that will automatically overwrite that empty location with the current data and set the initialization flag to true. If that location is addressed again, the compare would have to be performed. When the location is finally read and its color value is sent to the antialiaser, the initialization flag will be reset to false. The background cycle has no effect on the initialization flag. If a background cycle is used to initialize memory with sky data, all subsequent accesses to each pixel location will automatically write data using the short INIT cycle, thereby saving compares.

The next four cycles are compare cycles that occur when the stored depth value is read and compared with the incoming depth value. These compares must be executed in two parts because of the limited width of the available magnitude comparators. The depth word is 24 bits wide with 5 bits reserved for the exponent and 19 bits comprising the mantissa of the floating-point value. The magnitude of the exponent is compared first. Figure 12 illustrates the advantages of this approach. Since there are 5 bits to be compared, the field-of-view is effectively divided into five nonuniform regions of increasing depths. If the old pixel data is in a deeper region than the incoming pixel data based solely on exponent comparison, then a look-ahead-write cycle is initiated in which the new data is written over the old data without the additional comparison of the mantissas. If the old data is in a nearer region, then the controller initiates an ignore cycle in which it discards the incoming data without further comparisons. If the new and old depths are in the same region, then an additional compare on the mantissa must be executed to determine which pixel is visible. A read-compare-write cycle is initiated if the new pixel is determined to be closer, and its data is to replace what is currently stored. Otherwise, a read-compare-nowrite cycle is initiated in which the new data is ignored.

The remaining cycle is the memory read cycle. The antialiaser initiates this cycle by providing an address and securing the color data bus. While each color word is being read to the output bus, the associated INIT flag is being cleared.

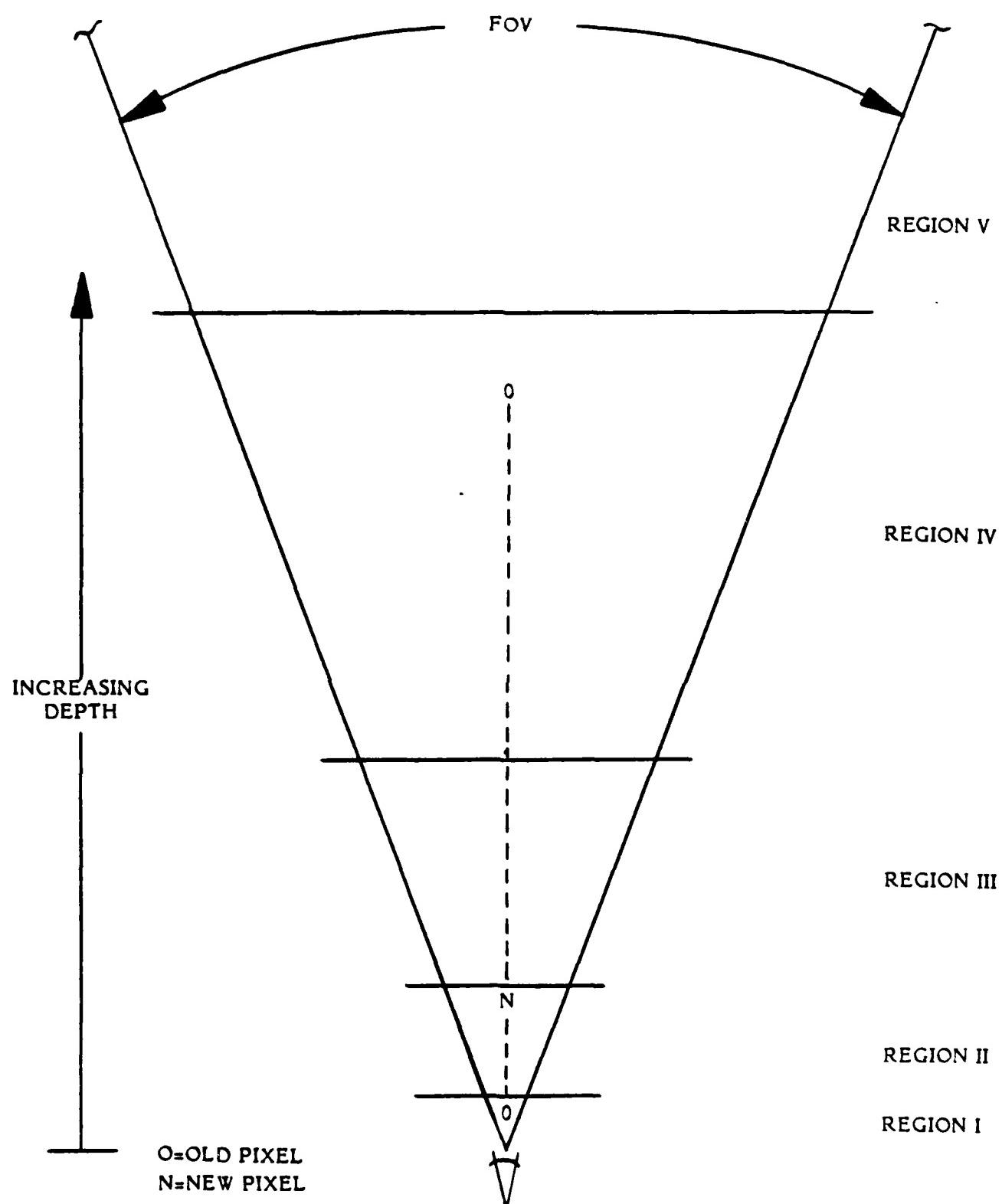
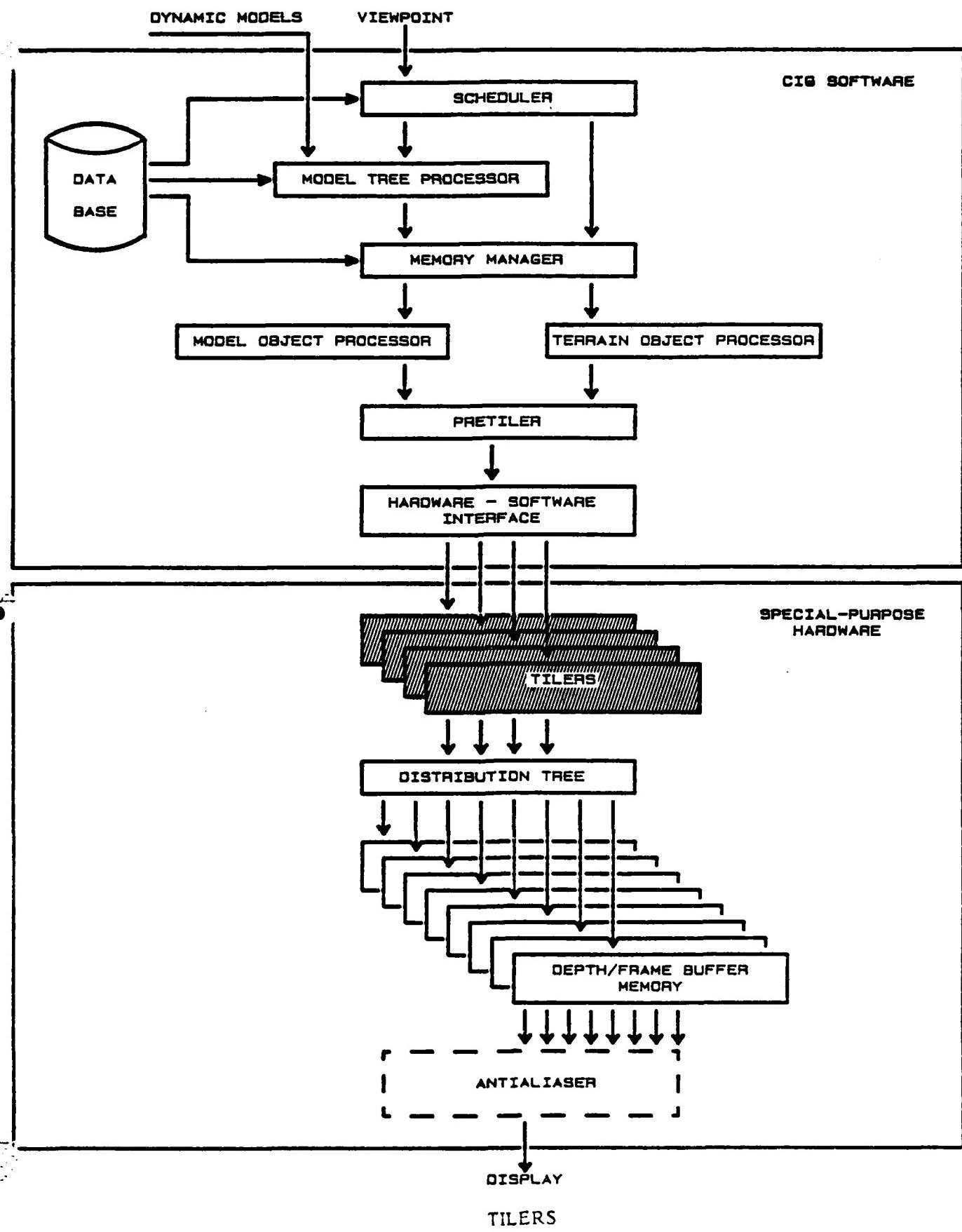


Figure 12. Depth Exponent Compare

4.1.4 FUTURE ENHANCEMENTS

Application of the depth/frame buffer memory to a real-time CIG system will require several enhancements. For instance, the color portion of the memory will be double-buffered for speed. Each depth/frame buffer memory will contain two color memory sections, one for displaying the current frame and the second for processing the next frame. Other improvements will be driven by technology trends such as the trend toward faster and denser memory components. Improvements in component technology will result in enhancements such as synchronous fixed-length cycles, simplified control logic, and lower parts count.



4.2 TILERS

4.2.1 BASIC CONCEPT

The tiler is the processor which renders shaded triangles for a depth-buffer CIG system. It is a pipelined, parallel-processing, special-purpose computer whose input is triangle information consisting of 1) screen coordinates of the three vertices and 2) parameters which define how color and depth vary across the interior of the triangle. The tiler must perform the following functions: 1) find which pixels lie within the triangle and 2) interpolate the depth and color between the vertices. The tiler output consists of the screen coordinates, depth, and color of each pixel.

4.2.2 ALGORITHM DEVELOPMENT

Software simulation of various tiling algorithms revealed several important issues. This section will summarize those issues and develop the tiling algorithm that was ultimately implemented.

4.2.2.1 REPROJECTION

The term reprojection refers to the process of projecting a point from viewpoint space to screen space, and then from screen space back to viewpoint space.

The concept of reprojection addresses the problem of incorrect prioritization of two closely spaced parallel planes, as illustrated in Figure 13. Because the pixel resolution is finite, point A on the bottom plane and point B on the top plane both project to pixel X on the screen. The depth-buffer algorithm would give viewing priority to the bottom plane since U_A is less than U_B . Obviously, this is an error. To calculate the proper depths, a line of sight projected ("reprojected") through the center of screen pixel X (dashed line in Figure 13) yields the proper depths U_C and U_D . Since U_C is less than U_D , the top and bottom planes are now correctly prioritized.

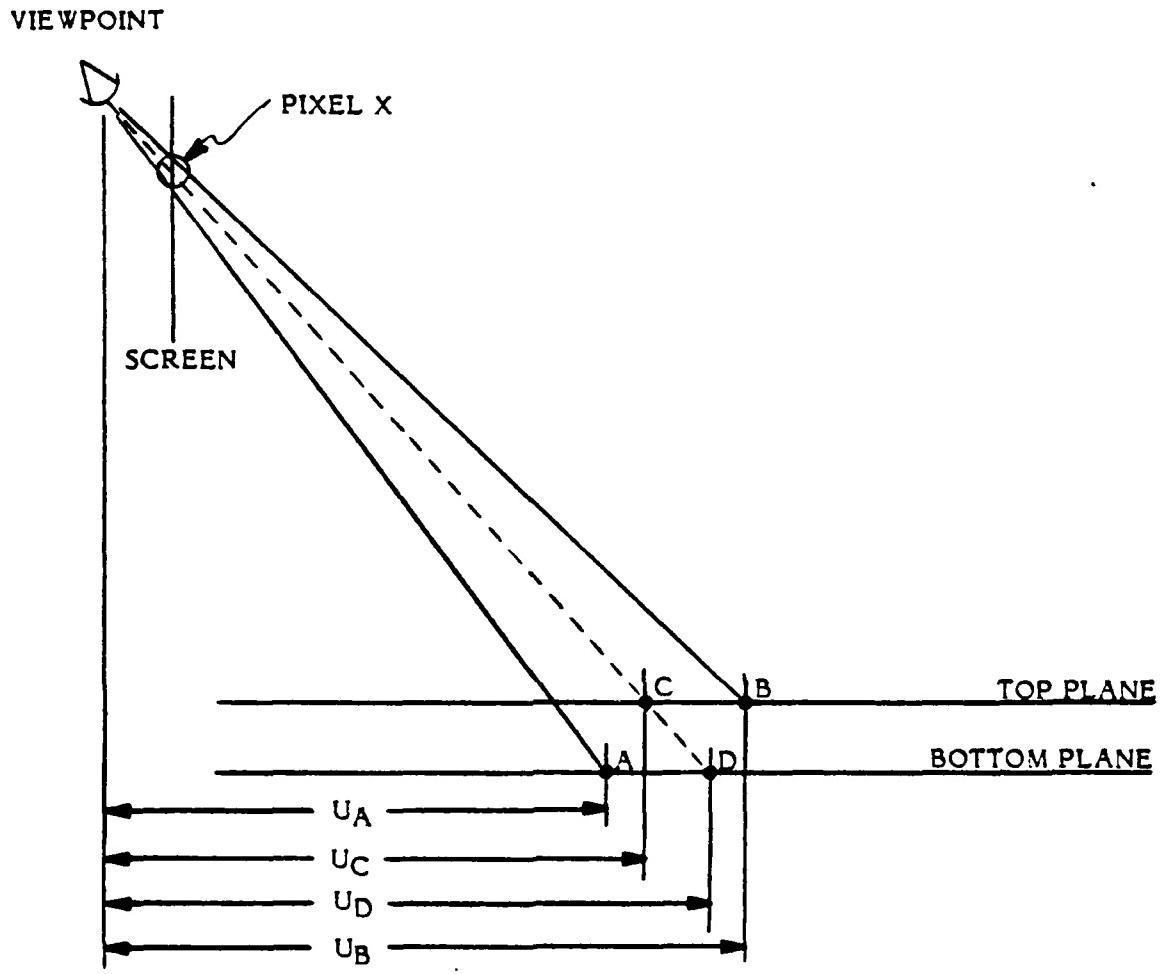


Figure 13. Reprojection

4.2.2.2 LINEARITY OF I/U AND C/U

The concept of reprojection presented in the previous section can be reformulated as follows: to guarantee correct prioritization, depths must be computed along a line of sight which is standardized for each pixel, e.g., through the center of the pixel. The vertical and horizontal screen coordinates are represented by i and j , respectively, and the center of each pixel is represented by the integral values of (i,j) . Computing the depth at the center of the pixel requires computing the depth for integral values of (i,j) . One therefore needs to find a closed-form expression for depth in terms of (i,j) .

Use the viewpoint coordinate U , which is normal to the screen, as a measure of depth. Since triangles define a planar surface, U is related to the other viewpoint coordinates through the equation of a plane:

$$AU + BV + CW = D$$

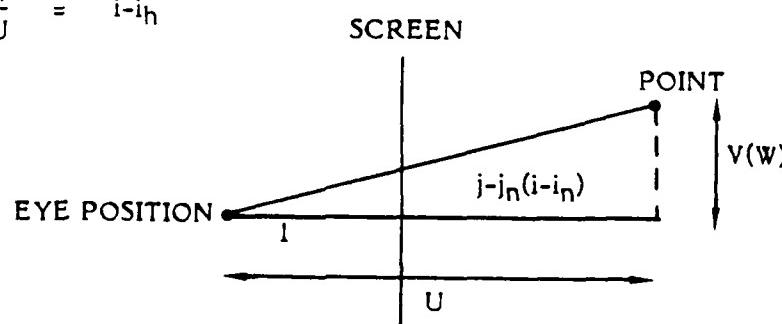
where (U,V,W) is a point in viewpoint space and
 A,B,C,D are the constants which define the plane

Along the line-of-sight through pixel (i,j) the viewpoint coordinates are related as follows:

$$\frac{V}{U} = j - j_h$$

NOTE: horizontal and vertical FOV scaling is included in the viewpoint coordinates

$$\frac{W}{U} = i - i_h$$



where (i_h, j_h) are the coordinates of the center of the screen.

Combine these equations to find how U depends on i and j :

$$\frac{A}{D} + \frac{B}{D} \left(\frac{V}{U} \right) + \frac{C}{D} \left(\frac{W}{U} \right) = \frac{1}{U}$$

$$\frac{1}{U} = \frac{A}{D} + \frac{B}{D} (j - j_h) + \frac{C}{D} (i - i_h)$$

This equation demonstrates that $1/U$, rather than U , depends linearly on i and j .

From this, it is apparent that $1/U$ is a more natural measure of depth than U in a system which processes perspective views of planar surfaces. The task of the tiler, which computes a depth for each pixel within the triangle, would be greatly simplified if depth were a linear function of i and j . However, we must also consider whether the depth resolution, ΔU , is worsened by using $1/U$ rather than U .

This question cannot be answered without considering whether the depth will be represented as a fixed-point or floating-point number. The results of this analysis appear in Table 2. $1/U$ in fixed-point representation clearly has unacceptable depth resolution for large U ; however, $1/U$ in floating-point representation has the same depth resolution as U in floating-point representation. Therefore, a floating-point representation of $1/U$ is chosen as the measure of depth.

Table 2. Depth Resolution ΔU for Different Representations of Depth

Depth	Representation	ΔU	Comments
U	Fixed point	$2^{-(F+1)}$	Limited range
	Floating point	$U 2^{-(M+1)}$	
$1/U$	Fixed point	$U^2 2^{-(F+1)}$	Natural for perspective views of planar
	Floating point	$U 2^{-(M+1)}$	surfaces; clean representation of $U=\infty$

Notes: F = number of bits of fraction in fixed-point representation
M = number of bits of mantissa in floating-point representation, not counting the high-order bit of the mantissa, which is assumed to be 1.

An analogous result holds for the dependence of color upon i and j . Color is expressed as the intensity of the three primary colors, (R,G,B), for red, green, and blue, respectively. In the Gouraud shading model, each intensity is assumed to vary linearly across the plane of the triangle. Each primary color intensity, c , can therefore be expressed as a linear function of the viewpoint coordinates:

$$c = EU + FV + GW + H$$

where E,F,G,H define how the intensity varies in viewpoint space.

To express the intensity in terms of screen coordinates, use the previous expression for $1/U$:

$$\begin{aligned} \frac{c}{U} &= E + F \frac{V}{U} + G \frac{W}{U} + H \cdot \frac{1}{U} \\ &= E + F(j-j_o) + G(i-i_o) + \\ &\quad H \left[\frac{A}{D} + \frac{B}{D}(j-j_h) + \frac{C}{D}(i-i_h) \right] \\ &= \left[E + \frac{AH}{D} \right] + \left[F + \frac{B}{D} \right](j-j_o) + \left[G + \frac{C}{D} \right](i-i_o) \end{aligned}$$

Therefore, this derivation demonstrates that c/U , not c , depends linearly upon i and j .

However, in the case of color, the "natural" form c/U cannot be used beyond the tiling process, since only the intensity c is meaningful. Ideally, c/U would be used in the tiling process with the true intensity, c , retrieved for each pixel by computing

$$c = \frac{c/U}{1/U}$$

However, since c/U has a much larger dynamic range than c , the computation of c/U rather than c would result in a substantial increase in cost and complexity of internal circuitry. Also, the division by $1/U$ to recover the color is difficult and

expensive. It was therefore decided to approximate the color c for each pixel by an interpolation which is linear in screen space:

$$c \cong c_0 + \frac{\partial c}{\partial i} (i - i_0) + \frac{\partial c}{\partial j} (j - j_0)$$

where c_0 , $\frac{\partial c}{\partial i}$, $\frac{\partial c}{\partial j}$ define how color varies across the screen.

The visual effect of this approximation is illustrated in Figure 14.

4.2.2.3 DEPTH AND COLOR INTERPOLATION

Interpolation of the depth and color between vertices to find these values at interior points requires division and multiplication. Since the cost of implementing these operations in hardware is high, there is a need to find a method that avoids division and multiplication. A description of such a method follows.

In the previous section an expression that shows the functional dependence of color upon screen coordinates was derived. Similarly, the derivation of the functional dependence of depth upon screen coordinates is as follows:

$$d = d_0 + \frac{\partial d}{\partial i} \Delta i + \frac{\partial d}{\partial j} \Delta j$$

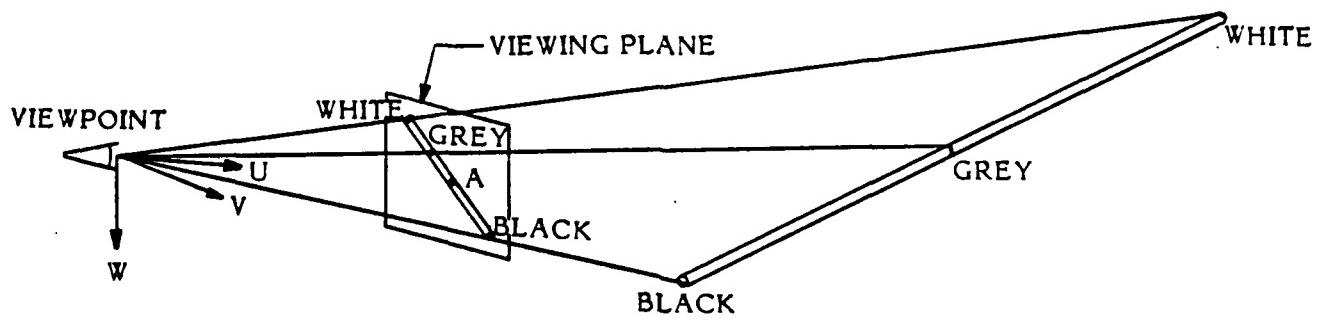
where $d = 1/U$ is the depth at (i, j) ;

d_0 is the depth at a triangle origin (i_0, j_0) ;

$\frac{\partial d}{\partial i}$, $\frac{\partial d}{\partial j}$ are the slopes of depth in the i and j direction, respectively;

and

$\Delta i = i - i_0$ $\Delta j = j - j_0$ are the displacement from the triangle origin.



The rod in the diagram is colored black at one end, white at the other, and is linearly shaded in the middle. The midpoint of the rod is grey. The diagram shows that the midpoint of the rod does not project to the middle of the rod's projection. Linear color interpolation in screen space would therefore lead to an apparent shift of the rod's midpoint.

Figure 14. Projected Color and Linearity

Define:

$$SDI = \frac{\partial d}{\partial i} \Delta i$$

$$SDJ = \frac{\partial d}{\partial j} \Delta j$$

$$SCI = \frac{\partial c}{\partial i} \Delta i$$

$$SCJ = \frac{\partial c}{\partial j} \Delta j$$

Rewrite depth and color expressions:

$$d = d_0 + SDI + SDJ$$

$$c = c_0 + SCI + SCJ$$

Given a known depth (color) at a pixel (i, j) , the depth (color) at pixel $(i, j \pm 1)$ is found by simply recalculating SDJ (SCJ):

$$SDJ \leftarrow SDJ \pm \frac{\partial d}{\partial j} \quad j = \pm 1$$

$$d = d_0 + SDI + SDJ$$

$$SCJ \leftarrow SCJ \pm \frac{\partial c}{\partial j}$$

$$c = c_0 + SCI + SCJ$$

The same applies to pixel $(i \pm 1, j)$

$$SDI \leftarrow SDI \pm \frac{\partial d}{\partial i} \quad i = \pm 1$$

$$d = d_0 + SDI + SDJ$$

$$SCI \leftarrow SCI \pm \frac{\partial c}{\partial i}$$

$$c = c_0 + SCI + SCJ$$

Although division is required to calculate the slopes of depth and color in the i and j direction, this is only done once. The actual depth and color interpolation could be implemented by using only additions and subtractions.

4.2.2.4 INTERIOR/EXTERIOR PIXEL DETERMINATION

Given the three vertices of a triangle in screen space, one method of computing the location of the triangle's edges is interpolation between the vertices. However, such an edge interpolation requires divisions and multiplications. The preceding subsection discussed a method of interpolating color and depth that could be implemented using only additions and subtractions if an algorithm were devised that samples the interior by moving through the triangle in increments of one horizontal (j) or vertical (i) pixel. This subsection discusses such an algorithm (see also (Cyrus78)).

Figure 15 shows two triangles in screen space with normals to the edges given. In both cases, calculating the dot products $\vec{V}_1 \cdot \vec{n}_1$, $\vec{V}_2 \cdot \vec{n}_2$, $\vec{V}_3 \cdot \vec{n}_3$ will determine if pixel (i,j) is interior or exterior to the triangle. For case 1, pixel (i,j) is interior to the triangle so the condition for a point inside the triangle is:

$$\vec{V}_1 \cdot \vec{n}_1 \geq 0 \text{ and } \vec{V}_2 \cdot \vec{n}_2 \geq 0 \text{ and } \vec{V}_3 \cdot \vec{n}_3 \geq 0$$

For case 2, pixel (i,j) is exterior to the triangle and since $\vec{V}_1 \cdot \vec{n}_1$ is less than 0, the condition for a point outside the triangle is:

$$\vec{V}_1 \cdot \vec{n}_1 < 0 \text{ or } \vec{V}_2 \cdot \vec{n}_2 < 0 \text{ or } \vec{V}_3 \cdot \vec{n}_3 < 0$$

The vectors, normals, and dot products are easily calculated:

$$\vec{V}_1 = (i - i_1, j - j_1)$$

$$\vec{V}_2 = (i - i_2, j - j_2)$$

$$\vec{V}_3 = (i - i_3, j - j_3)$$

$$\vec{n}_1 = (n_{1i}, n_{1j}) = (j_3 - j_1, i_1 - i_3)$$

$$\vec{n}_2 = (n_{2i}, n_{2j}) = (j_1 - j_2, i_2 - i_1)$$

$$\vec{n}_3 = (n_{3i}, n_{3j}) = (j_2 - j_3, i_3 - i_2)$$

$$\vec{P}_1(i,j) = [(i - i_1)(j_3 - j_1) + (j - j_1)(i_1 - i_3)]$$

$$\vec{P}_2(i,j) = [(i - i_2)(j_1 - j_2) + (j - j_2)(i_2 - i_1)]$$

$$\vec{P}_3(i,j) = [(i - i_3)(j_2 - j_3) + (j - j_3)(i_3 - i_2)]$$

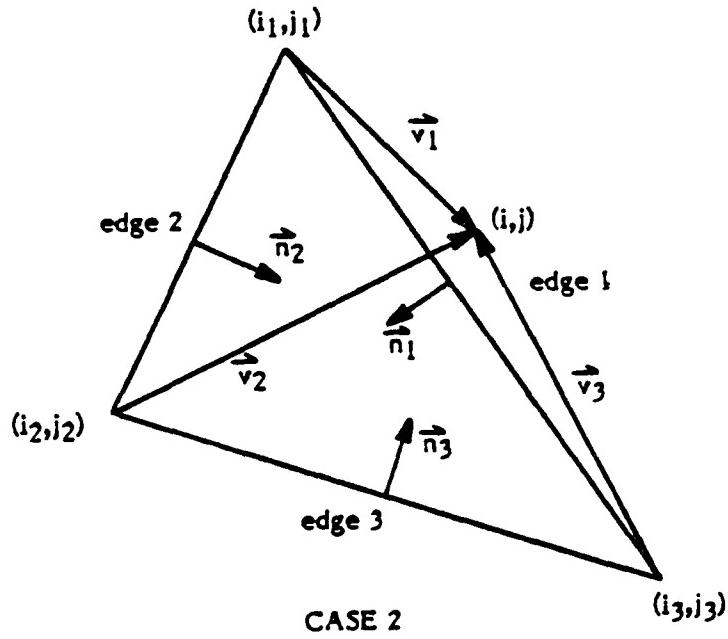
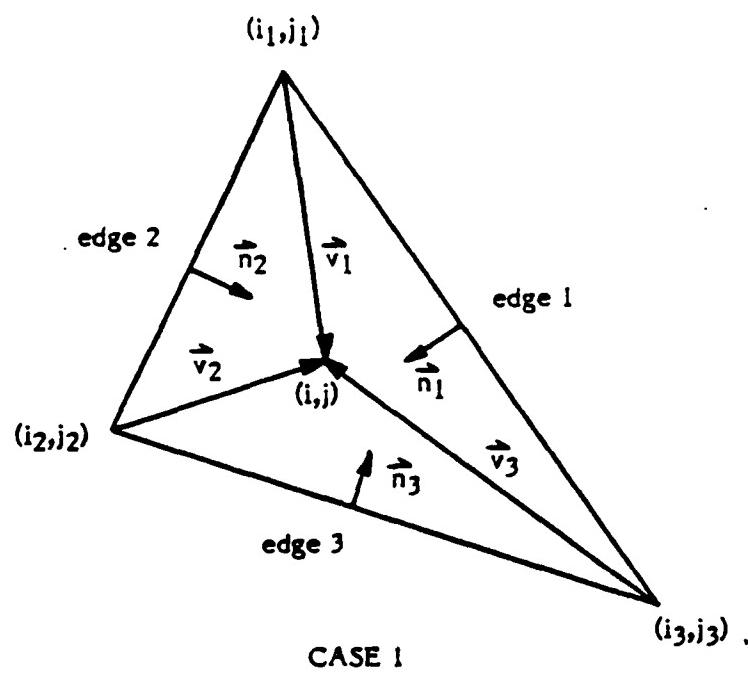


Figure 15. Dot Product Test

From the expression for $P_1(i,j)$, it is apparent that the dot product $P_1(i \pm 1, j)$ can be calculated recursively:

$$P_1(i \pm 1, j) = ((i \pm 1) - i_1)(j_3 - j_1) + (j - j_1)(i_1 - i_3) = P_1(i, j) \pm n_1 i$$

The dot product increments going from pixel (i, j) to $(i \pm 1, j)$ or $(i, j \pm 1)$ are calculated analogous to the depth and color calculations described in the preceding subsection:

$$P_k(i \pm 1, j) = P_k(i, j) \pm n_k i$$

$$P_k(i, j \pm 1) = P_k(i, j) \pm n_k j$$

where $k = 1, 2, 3$

The initial dot product must be calculated at some fixed pixel and although multiplication is necessary, it is only done once for each triangle. The dot product for any other pixel is calculated by a succession of additions and/or subtractions.

4.2.2.5 OVERSAMPLING AND UNDERSAMPLING

Implementing a dot product test precisely as described introduces sampling problems on the triangle boundaries. Example 1 of Figure 16 shows the interior pixels of a narrow triangle. This case, called undersampling, demonstrates that it is possible to have raster lines through a triangle with no pixels internal to the triangle. In cases where this is unacceptable, a tolerance in the test of boundary pixels is required. The precise dot product test for some pixel (i, j) is:

$$P_1(i, j) \geq 0 \text{ and } P_2(i, j) \geq 0 \text{ and } P_3(i, j) \geq 0$$

The dot product test can be adjusted by incorporating tolerances into the inequalities:

$$P_1(i, j) \geq -t_1 \text{ and } P_2(i, j) \geq -t_2 \text{ and } P_3(i, j) \geq -t_3$$

where t_1, t_2, t_3 are positive tolerances;

or: $P_1(i, j) + t_1 \geq 0 \text{ and } P_2(i, j) + t_2 \geq 0 \text{ and } P_3(i, j) + t_3 \geq 0$

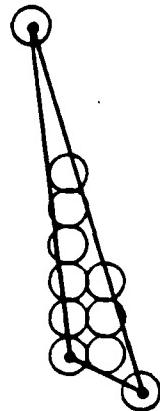
A good tolerance choice would cause an edge pixel to be included if the edge passes within half a pixel's width of the pixel's center:

$$t_1 = 1/2 \text{ MAX} (|n_{1i}|, |n_{1j}|)$$

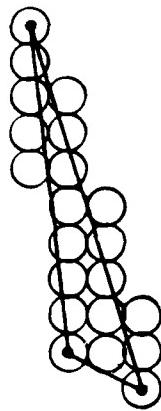
$$t_2 = 1/2 \text{ MAX} (|n_{2i}|, |n_{2j}|)$$

$$t_3 = 1/2 \text{ MAX} (|n_{3i}|, |n_{3j}|)$$

Example 2 of Figure 16 shows the interior pixels of the same narrow triangle when the dot product test is adjusted. This case is called oversampling.



EXAMPLE 1 - UNDERSAMPLED



EXAMPLE 2 - OVERSAMPLED

Figure 16. Undersampling and Oversampling

Software simulations of the dot product tests revealed difficulties associated with both oversampling and undersampling.

In the oversampling case, edge pixels of adjoining triangles will overlap. This will increase the depth complexity of the picture. Figure 17 illustrates another problem caused by oversampling. If plane A is oversampled, some of the boundary pixels will "poke through" plane B which is perpendicular to plane A. This effect is quite noticeable when plane A and plane B are of different colors.

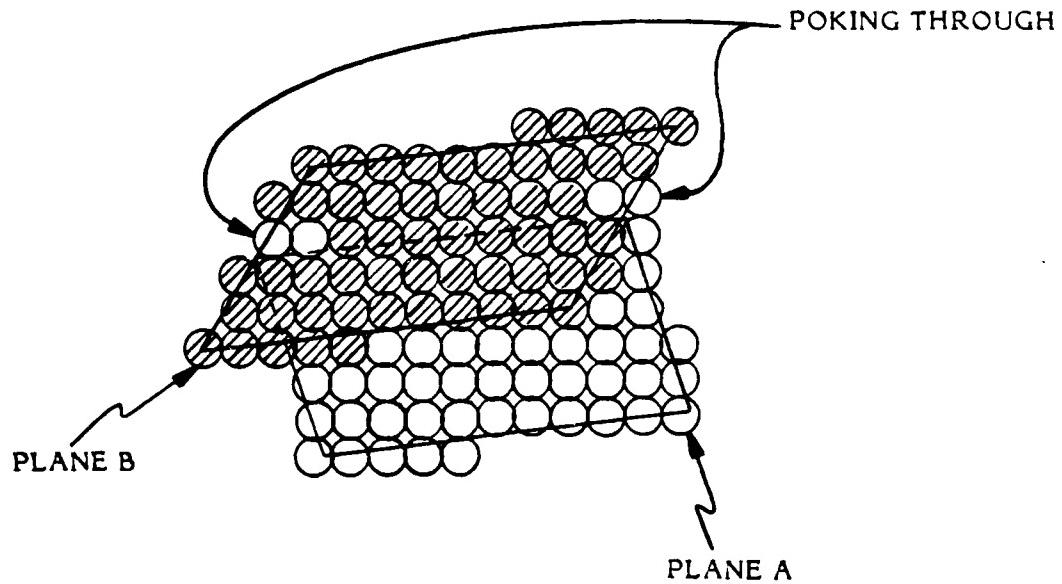


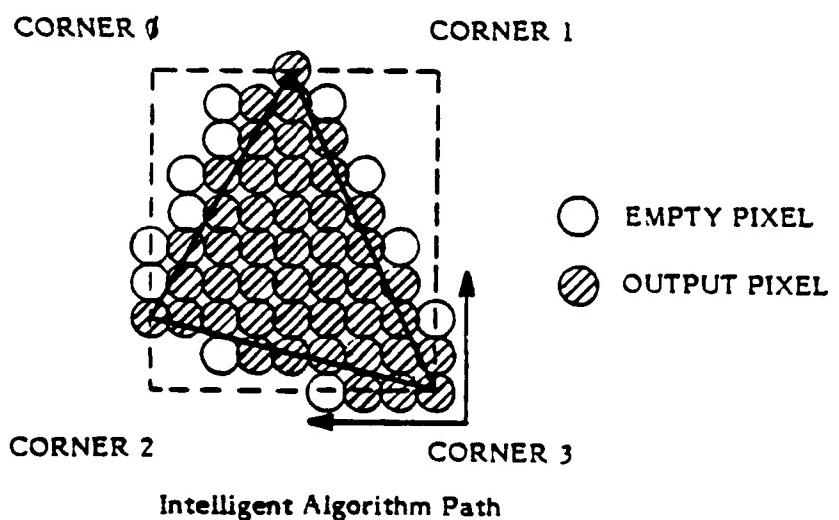
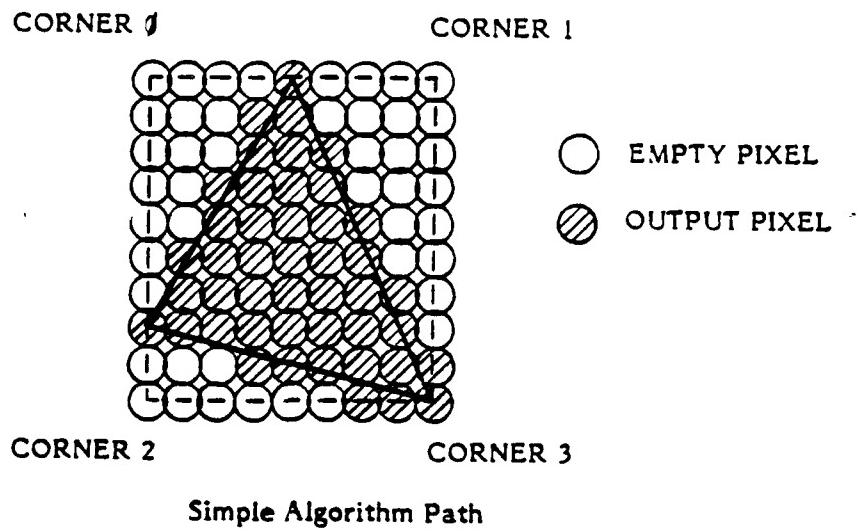
Figure 17. "Poking Through" Effect

As previously discussed, undersampling essentially sets the tolerances equal to zero. However, analysis of anomalies in pictures created from undersampled triangles indicate that the tolerances should not be set equal to zero. Instead, they should be set to some small value to compensate for round-off errors. The expressions for undersampling tolerances will be given in subsection 4.2.2.7.

It is apparent that a hardware tiler which can oversample as well as undersample is required. The majority of triangles will be undersampled, avoiding the increase in depth complexity and the "poking through" effect. When undersampled triangles become visually distracting because of unconnected pixels, oversampling will be selected.

4.2.2.6 CONTROL OF SAMPLING PATH

At this point, a tiler design has emerged where the dot products, depth, and color are computed by combining values at some fixed point on the triangle with increments in the i and j directions. Consider the triangle shown in Figure 18 and the rectangular box circumscribing it. The initial dot products, depth, and color



Note: Both examples are oversampled.

Figure 18. Sampling Paths

could be specified at corner 0, 1, 2, or 3. A simple algorithm would then sample each pixel inside the box and output those pixels inside the triangle. Obviously this type of implementation wastes considerable time sampling empty pixels. Therefore an algorithm has been developed that intelligently controls the sampling path to substantially reduce the time spent sampling empty pixels.

The key to the intelligent algorithm is choosing the correct starting point. Two issues are apparent: 1) any triangle can be circumscribed by a rectangular box and 2) at least one vertex of the triangle will also be a corner of the box. For the triangle in Figure 18 the correct starting point is the vertex corresponding to corner 3.

The basic concepts behind the intelligent tiling algorithm are simple. Sampling proceeds from the starting point incrementing horizontally until the dot product test flags an empty pixel. The starting point of each horizontal line is saved and then incremented vertically to form the starting point of the next line. The sampling path is always bounded by the circumscribing box. Figure 18 shows the sampling path of the intelligent algorithm.

A flag passed to the tiler designates the triangle vertex to be used as the starting point. The next subsection will detail the information input to the tiler, all setup calculations, and the intelligent algorithm.

4.2.2.7 INTELLIGENT TILING ALGORITHM

This subsection will complete the algorithm development discussions. Triangle variables input to the tiler are shown in Figure 19 and Table 3 defines each variable.

Before the actual tiling process can begin, a number of setup calculations must be made. The section of the tiler that performs these calculations is called the tiling machine setup. Table 4 defines the variables used in this section and Table 5 lists all the required calculations.

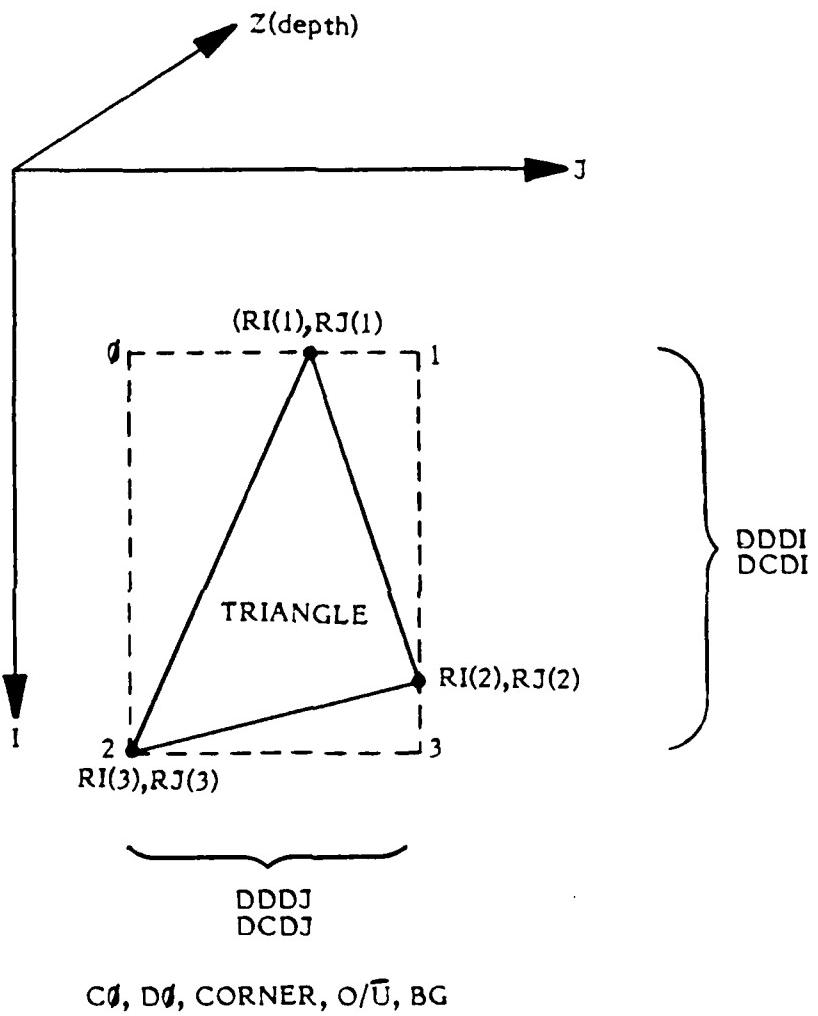


Figure 19. Triangle Variables

Table 3. Triangle Variable Definition

RI(n)	- I component of the vertex. Note that the three vertices (n=1,2,3) are ordered in the I direction.
RJ(n)	- J component of the vertex.
CORNER	- Flag indicating the vertex at which tiling begins. CORNER = 0,1,2,3 and corresponds to a corner of the circumscribing box.
O/U	- Mode flag setting the tiler for either undersample or oversample operation.
BG	- Mode flag defining a triangle as background or foreground.
C0	- Color of the CORNER vertex.
D0	- Depth of the CORNER vertex.
DDDI	- Change in depth per unit change in I ($\partial d / \partial i$)
DCDI	- Change in color per unit change in I ($\partial c / \partial i$)
DDDJ	- Change in depth per unit change in J ($\partial d / \partial j$)
DCDJ	- Change in color per unit change in J ($\partial c / \partial j$)

Table 4. Tiling Machine Setup Variables

NI(n)	- I component of the normal vector.
NJ(n)	- J component of the normal vector.
PTEST	- Test to determine if the normal vectors are pointing into or out of the triangle.
ISTART	- I component of the first pixel tested for a given line. Initialized to the I component of the CORNER vertex.
ISTOP	- I component of the vertex at which tiling stops.
JSTART	- J component of the first pixel tested for a given line. Initialized to the J component of the CORNER vertex.
JSTOP	- J component of the last pixel in a line if the tiling of that line was not terminated by the dot product test.
PSTART(n)	- The initial dot product at the CORNER vertex.
PTOL(n)	- Tolerance factor, undersample or oversample.

Table 5. Tiling Machine Setup Calculations
 (Continued on Following Page)

IIMIN = RI(1)

IIMAX = RI(3)

JJMIN = MIN [RJ(1), RJ(2), RJ(3)]

JJMAX = MAX [RJ(1), RJ(2), RJ(3)]

NI(1) = RJ(3) - RJ(1)

NI(2) = RJ(1) - RJ(2)

NI(3) = RJ(2) - RJ(3)

NJ(1) = RI(1) - RI(3)

NJ(2) = RI(2) - RI(1)

NJ(3) = RI(3) - RI(2)

PTEST = NJ(1) · NI(3) - NI(1) · NJ(3)

IF PTEST is negative then:

NI(1) = -NI(1) NJ(1) = -NJ(1)

NI(2) = -NI(2) NJ(2) = -NJ(2)

NI(3) = -NI(3) NJ(3) = -NJ(3)

IF CORNER = 0

ISTART = IIMIN

ISTOP = IIMAX

JSTART = JJMIN

JSTOP = JJMAX

IF CORNER = 1

ISTART = IIMIN

ISTOP = IIMAX

JSTART = JJMAX

JSTOP = JJMIN

IF CORNER = 2

ISTART = IIMAX

ISTOP = IIMIN

JSTART = JJMIN

JSTOP = JJMAX

Table 5. Tiling Machine Setup Calculations (Concluded)

```

IF CORNER = 3
    ISTART = IIMAX
    ISTOP   = IIMIN
    JSTART = JJMAX
    JSTOP   = JJMIN

IF OVERSAMPLE
    PTOL(1) = 0.5 · MAX [ |NI(1)|, |NJ(1)| ]
    PTOL(2) = 0.5 · MAX [ |NI(2)|, |NJ(2)| ]
    PTOL(3) = 0.5 · MAX [ |NI(3)|, |NJ(3)| ]

IF UNDERSAMPLE
    PTOL(1) = [ 2-7(|NI(1)| + |NJ(1)|) +
                2-6(|(ISTART - RI(1))| + |(JSTART - RJ(1))|) ]
    PTOL(2) = [ 2-7(|NI(2)| + |NJ(2)|) +
                2-6(|(ISTART - RI(2))| + |(JSTART - RJ(2))|) ]
    PTOL(3) = [ 2-7(|NI(3)| + |NJ(3)|) +
                2-6(|(ISTART - RI(3))| + |(JSTART - RJ(3))|) ]

PSTART(1) = [(ISTART - RI(1))(NI(1))] +
            [(JSTART - RJ(1))(NJ(1))] +
            PTOL(1)
PSTART(2) = [(ISTART - RI(2))(NI(2))] +
            [(JSTART - RJ(2))(NJ(2))] +
            PTOL(2)
PSTART(3) = [(ISTART - RI(3))(NI(3))] +
            [(JSTART - RJ(3))(NJ(3))] +
            PTOL(3)

```

The section of the tiler that implements the intelligent tiling algorithm is called the tiling machine. Table 6 lists the variables that are input to the tiling machine from the tiling machine setup section. Table 7 defines the variables internal to the tiling machine.

Table 6. Tiling Machine Input Variables

PSTART(1)
NI(1)
NJ(1)
PSTART(2)
NI(2)
NJ(2)
PSTART(3)
NI(3)
NJ(3)
ISTART
ISTOP
JSTART
JSTOP
D0
DDDI
DDDJ
C0
DCDI
DCDJ
CORNER
BG

Table 7. Tiling Machine Internal Variables

P(n)	- The resultant dot product for a pixel being tiled. Initialized to PSTART(n).
E0	- Valid pixel flag. This 1 bit flag tags each pixel tiled as valid (within the triangle) or invalid (outside the triangle). E0 is true if $P(1) \geq 0$ and $P(2) \geq 0$ and $P(3) \geq 0$
E1	- Previous pixel valid flag. This 1 bit flag indicates that the previous pixel was valid or invalid. $E1(t) = E0(t-1)$
I	- I coordinate of a pixel tiled.
J	- J coordinate of a pixel tiled.
SDI	- Depth displacement along I axis.
SCI	- Color displacement along I axis.
SDJ	- Depth displacement along J axis.
SCJ	- Color displacement along J axis
SDJSAVE	- Temporary storage variable for SDJ.
SCJSAVE	- Temporary storage variable for SCJ.
PSAVE(n)	- Temporary storage variable for P(n).

Figure 20 shows a detailed flowchart of the intelligent tiling algorithm. Note that the algorithm is highly recursive and all operations are additions and/or subtractions.

The final step in the tiling process requires a summation of the depth and color partials. This section of the tiler is called tile accumulate. If the pixel output by the tiling machine was valid, the final depth and color calculations are made:

$$\begin{aligned} \text{Depth} &= D0 + SDI + SDJ \\ \text{Color} &= C0 + SCI + SCJ \end{aligned}$$

BG, I, J, Depth, Color are then output as a pixel from the tiler. If the pixel was invalid, then that pixel is discarded.

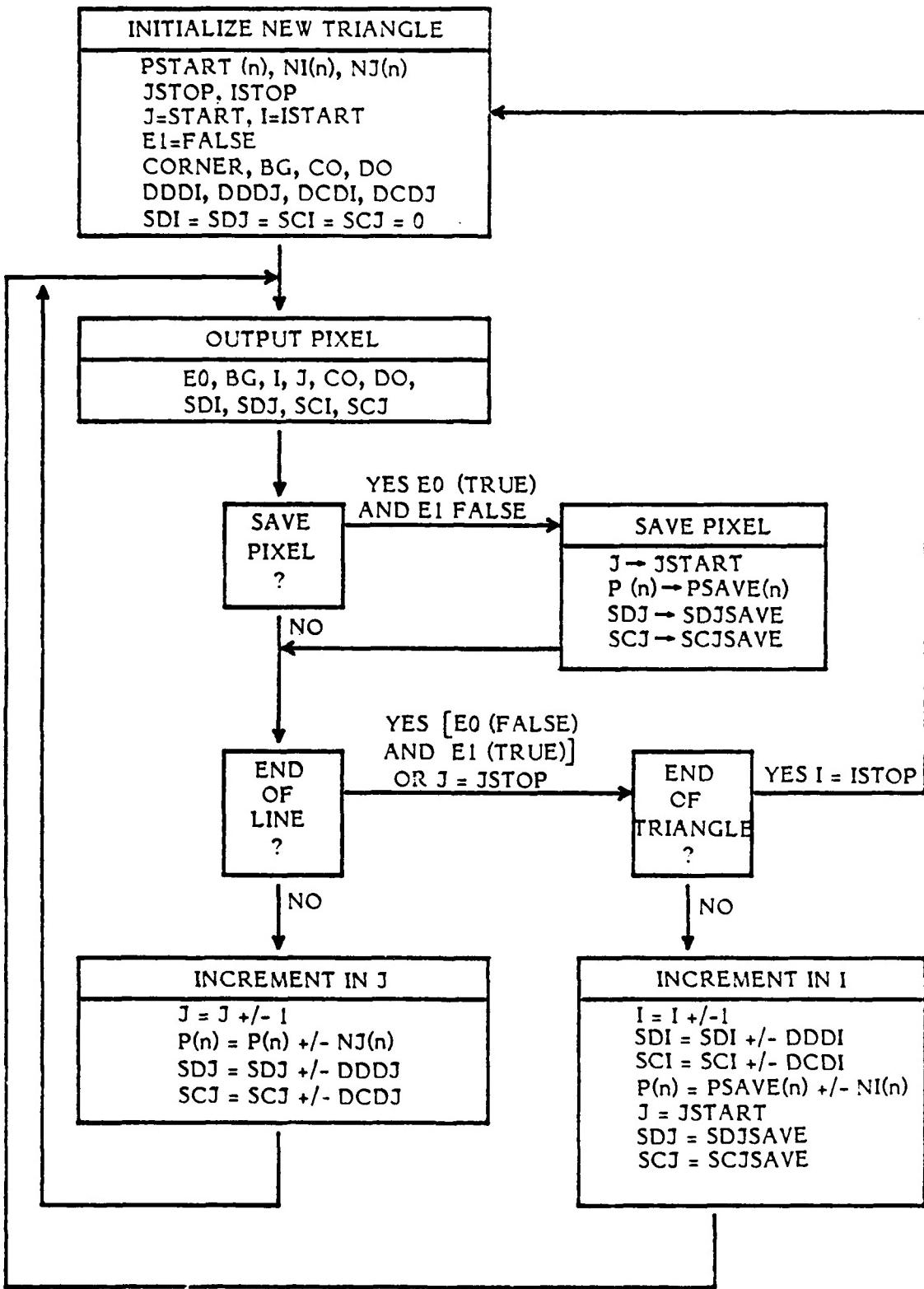


Figure 20. Intelligent Tiling Algorithm

4.2.3 HARDWARE DESIGN OBJECTIVES

Early in the development of the tiler, several important objectives were established. These were derived from analysis, software simulations, and system performance goals. This section will briefly discuss those objectives.

4.2.3.1 NUMERICAL ACCURACY

Sufficient numerical accuracy in critical calculations had to be provided, but costly overdesign had to be avoided. The major area of concern involved the calculations of the bounding vectors, normals, and dot products. It is possible to cause a variation of the "poking through" effect by rounding the vertex coordinates to integral values before the bounding vectors are calculated. The obvious solution is to calculate the bounding vectors and normals from the projected vertex coordinates before rounding. This of course requires additional precision for the dot products and normals. The additional precision was provided in order to sample the triangles as correctly as possible. Table 8 lists the precision and format of the various tiler variables.

4.2.3.2 PERFORMANCE

There are two parameters which describe the performance of the hardware tiler. The first parameter is the number of triangles that can be processed per second. The second parameter is the pixel output rate.

The bandwidth limitation of the host computer data channel connected to the hardware tiler is approximately 10 microseconds per triangle. Therefore, a performance goal of 100,000 triangles per second was established.

The system performance goal at the pixel level was established at 10 million pixels per second per tiler. This establishes the tiler output rate at 10 million pixels per second. An important implication of these two parameters is that the 100,000 triangles per second goal is unachievable if the average size of the triangles exceeds 100 pixels. The actual performance measurements of the hardware tiler appear in Section 7.0.

Table 8. Tiler Variable Precision and Format

VARIABLE	# BITS	PRECISION AND FORMAT								
RI(n), RJ(n) NI(n), NJ(n)	16	<table border="1"> <tr> <td>S</td> <td>FIXED</td> <td>POINT</td> </tr> <tr> <td>15 14</td> <td></td> <td>6 5 0</td> </tr> </table>	S	FIXED	POINT	15 14		6 5 0		
S	FIXED	POINT								
15 14		6 5 0								
I, J	9	<table border="1"> <tr> <td colspan="2">INTEGER</td> </tr> <tr> <td>8</td> <td>0</td> </tr> </table>	INTEGER		8	0				
INTEGER										
8	0									
P(n)	28	<table border="1"> <tr> <td>S</td> <td>FIXED</td> <td>POINT</td> </tr> <tr> <td>27 26</td> <td></td> <td>8 7 0</td> </tr> </table>	S	FIXED	POINT	27 26		8 7 0		
S	FIXED	POINT								
27 26		8 7 0								
CO, DCDI, DCDJ, SCI, SCJ	16	<table border="1"> <tr> <td>S</td> <td>EXP</td> <td>S</td> <td>MANTISSA</td> </tr> <tr> <td>15 14</td> <td></td> <td>11 10 9</td> <td>0</td> </tr> </table>	S	EXP	S	MANTISSA	15 14		11 10 9	0
S	EXP	S	MANTISSA							
15 14		11 10 9	0							
DO, DODI, DODJ, DSI, SDJ	32	<table border="1"> <tr> <td>S</td> <td>EXP</td> <td>S</td> <td>MANTISSA</td> </tr> <tr> <td>31 30</td> <td></td> <td>25 24 23</td> <td>0</td> </tr> </table>	S	EXP	S	MANTISSA	31 30		25 24 23	0
S	EXP	S	MANTISSA							
31 30		25 24 23	0							
DEPTH	24	<table border="1"> <tr> <td colspan="2">EXP</td> <td>MANTISSA</td> </tr> <tr> <td>23</td> <td>19</td> <td>18</td> </tr> </table>	EXP		MANTISSA	23	19	18		
EXP		MANTISSA								
23	19	18								
COLOR	8	<table border="1"> <tr> <td colspan="2">INTEGER</td> </tr> <tr> <td>7</td> <td>0</td> </tr> </table>	INTEGER		7	0				
INTEGER										
7	0									

4.2.3.3 MECHANICAL/ELECTRICAL

In considering the mechanical and electrical design of the tiler, the circuitry could occupy no more space than two wirewrap cards, approximately 14 by 17 inches each, and consume a maximum of 200 watts of 5 VDC power. Other than these specifications, there were few constraints placed on these characteristics of the hardware tiler.

4.2.4 HARDWARE DESIGN OVERVIEW

Figure 21 shows a top level block diagram of the hardware tiler functional blocks. This section will provide an overview of each block.

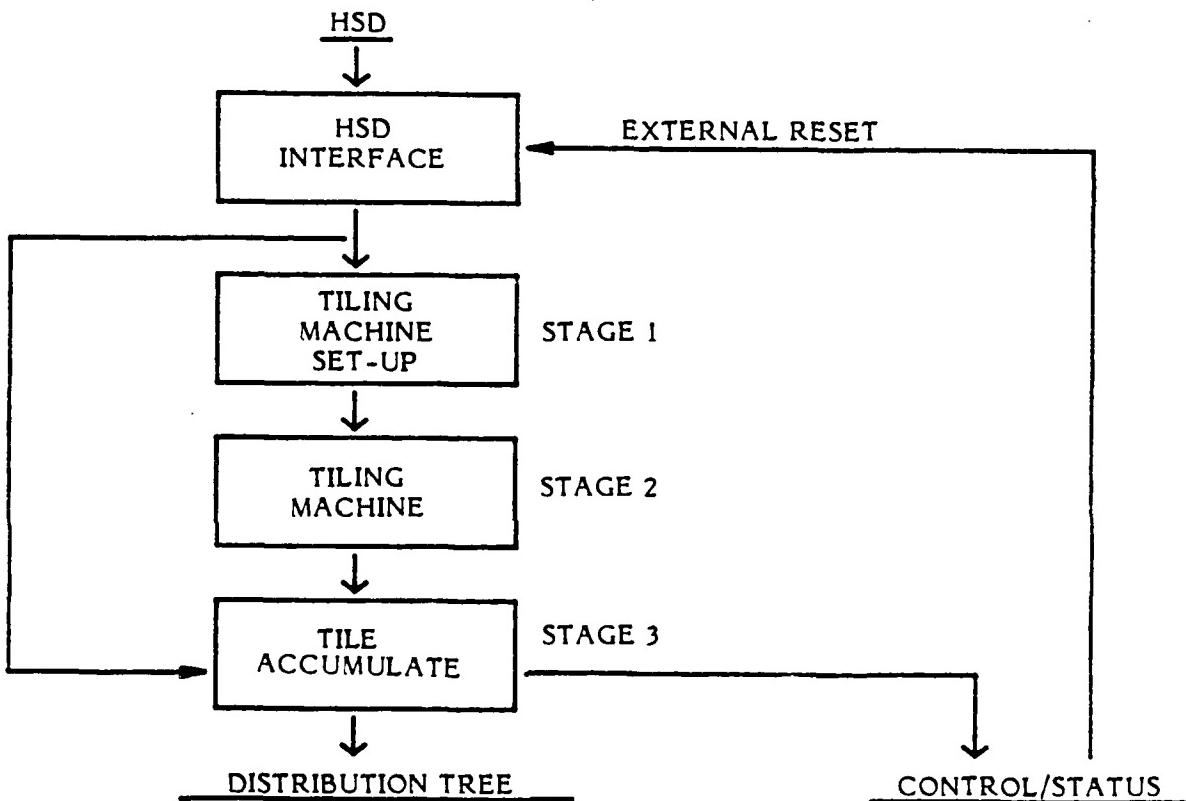


Figure 21. Tiler Functional Blocks

4.2.4.1 HSD INTERFACE

The HSD interface shown in Figure 22 provides the means for transferring triangle data from the SEL host computer to the hardware tiler. A total of eight 32-bit transfers is required for each triangle. Other features are the ability to transfer pixel data directly from the SEL host computer and the two-triangle-deep FIFO queue.

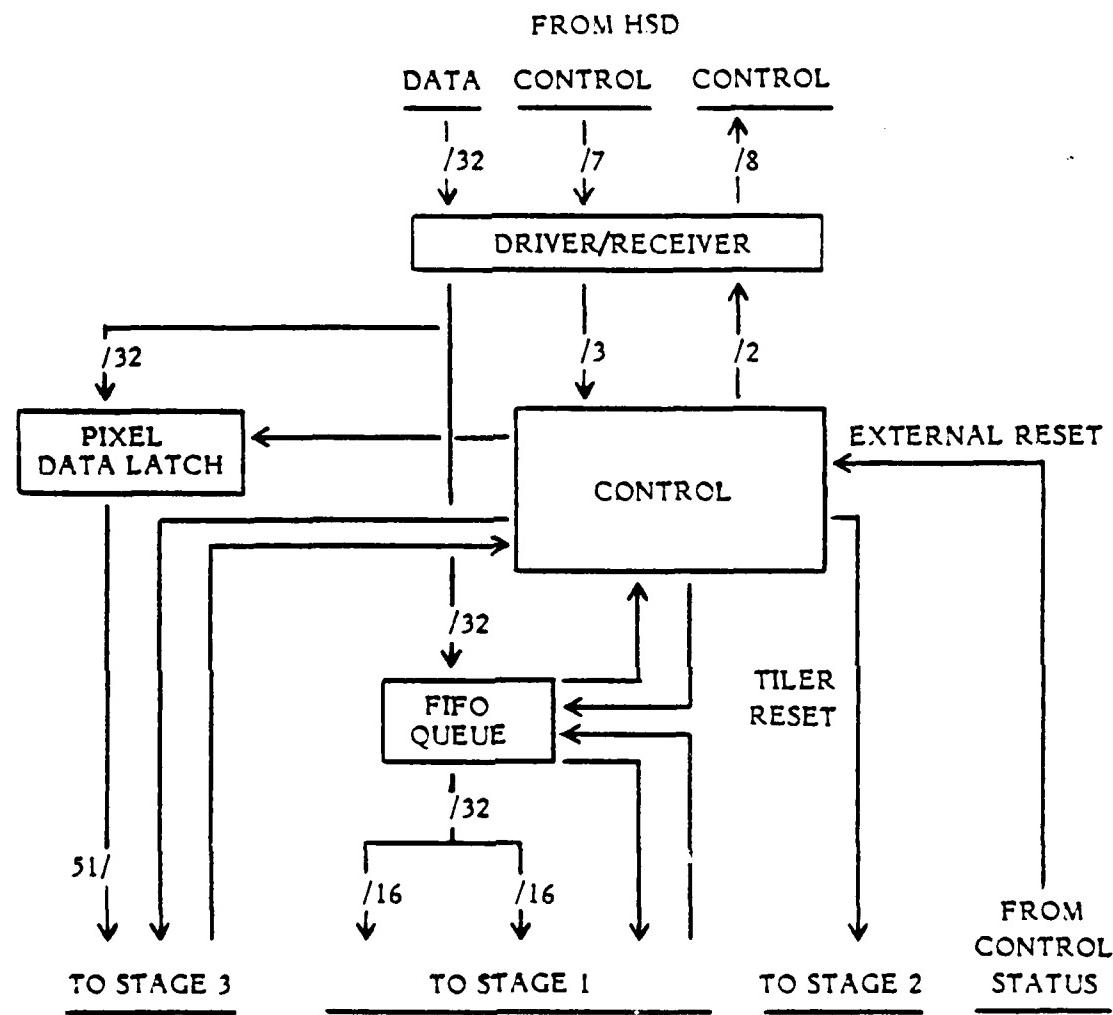


Figure 22. HSD Interface

4.2.4.2 TILING MACHINE SETUP

Subsection 4.2.2.7 lists a series of calculations that must be performed by the tiling machine setup section of the hardware tiler. A microprogrammable machine, shown in Figure 23, was developed to meet those requirements. The basic machine cycles at 8 MHz, utilizes two 16-bit microprocessors operating in parallel, and has a 16x16 bit hardware multiplier/accumulator. The microcode prepared for this machine was structured to match the 100,000 triangle per second performance goal. A five-triangle-deep FIFO queue buffers the data transfer between this section and the tiling machine.

4.2.4.3 TILING MACHINE

Figure 24 shows a block diagram of the tiling machine which directly implements the tiling algorithm discussed in subsection 4.2.2.7. The tiling machine controller is a finite-state machine that controls initialization and the tiling path. The basic cycle time of 10 MHz means that a pixel is tested and output every 100 ns whether it is valid or invalid. The dot product, i,j, depth, and color calculations are implemented with high-speed arithmetic logic units. The depth and color outputs (SDI, SDJ, SCI, and SCJ) are calculated in a fixed-point format and then converted to a normalized floating-point value before being output to the tile accumulate section.

4.2.4.4 TILE ACCUMULATE

The tile accumulate section represents the final step in the tiling process. The control block, shown in Figure 25, controls the basic timing of the tiling machine and discards invalid pixels. The depth and color summations require two floating-point adds each, which must be pipelined to maintain the 10 million pixel per second rate. The resulting depth value is reformatted to match the memory format and the resulting color value is converted from floating point to an integer. A 64-pixel-deep FIFO queue is provided to buffer the data transfer between the tiler and the distribution tree.

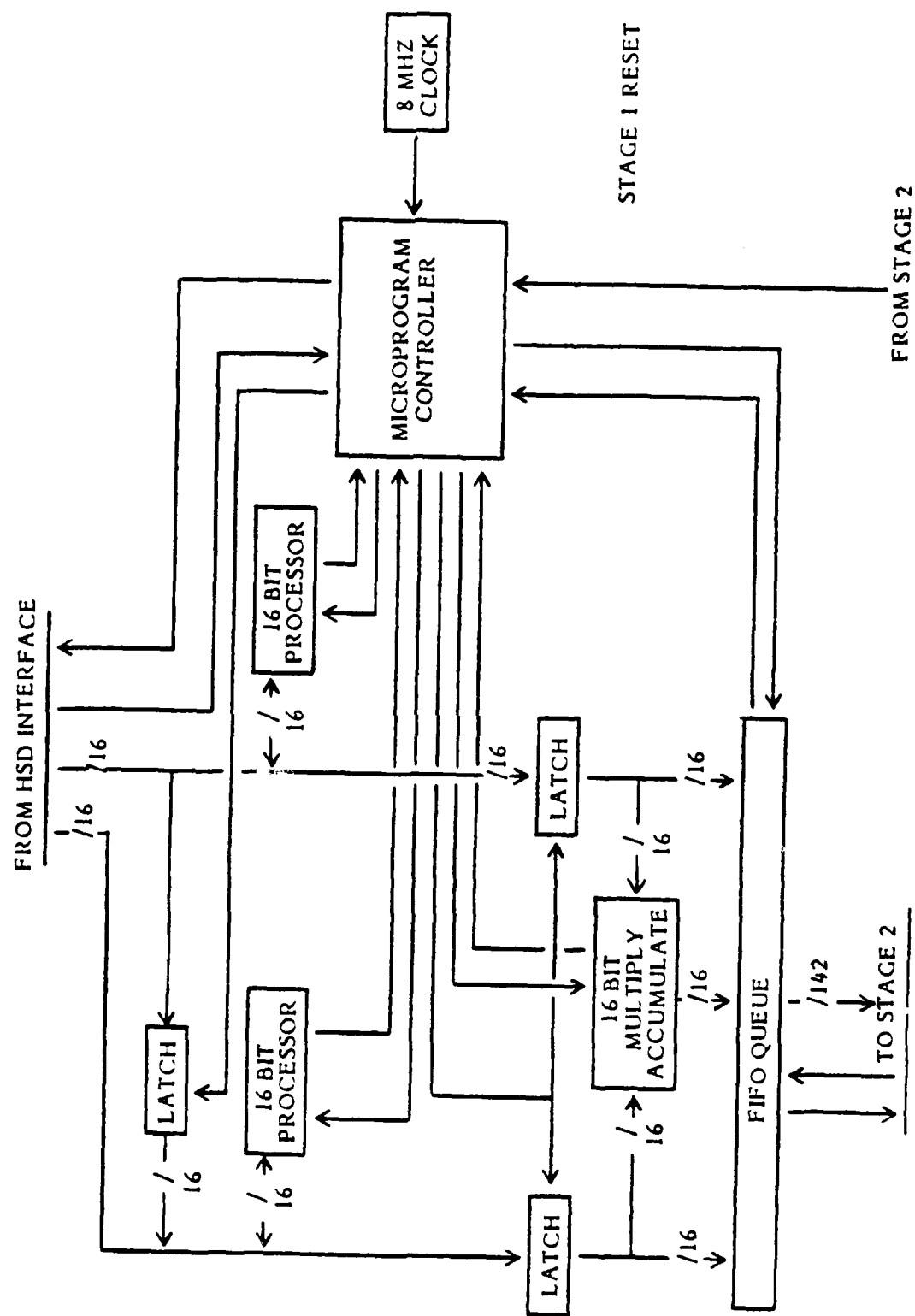


Figure 23. Tiling Machine Setup

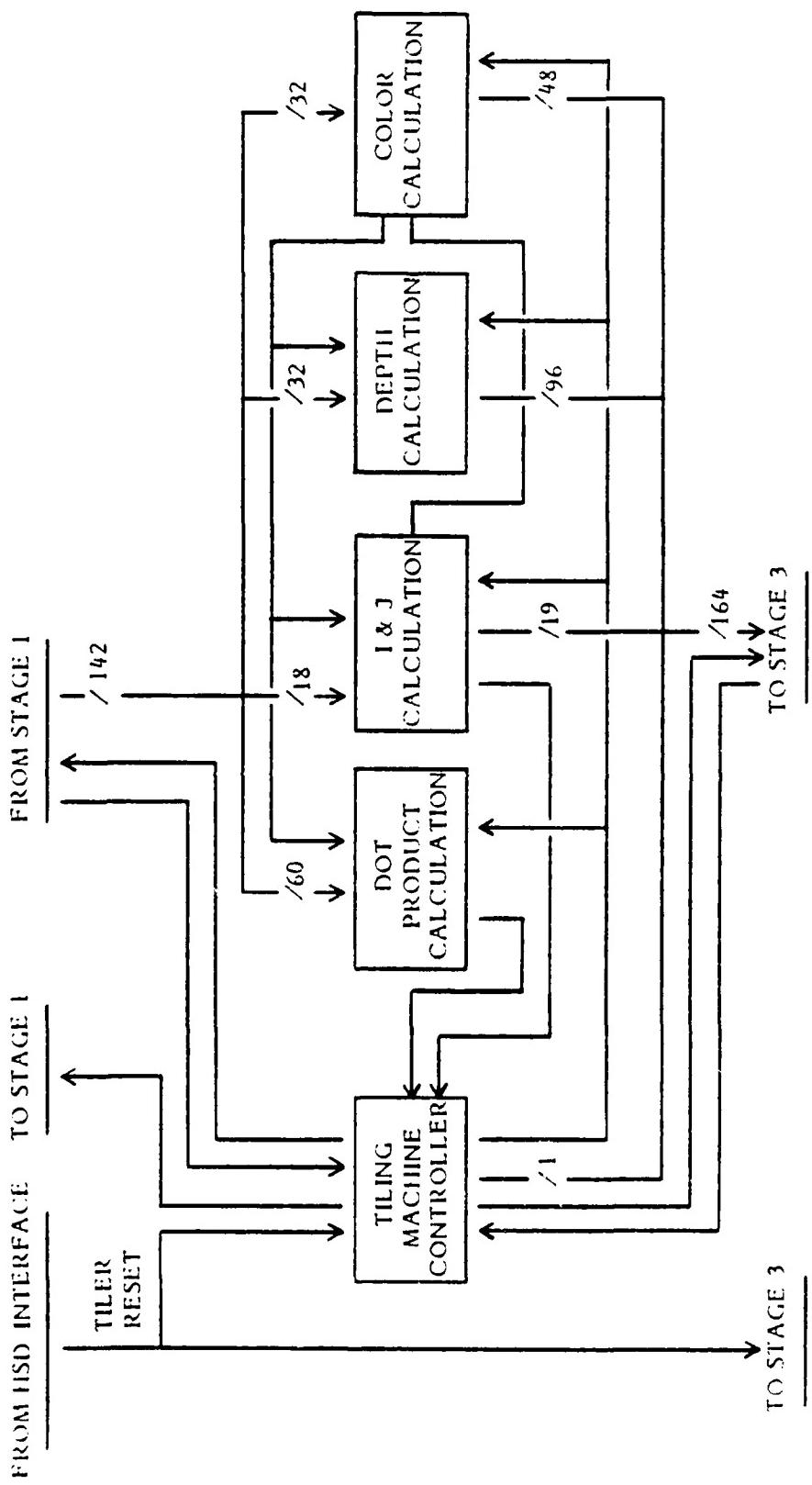


Figure 24. Tiling Machine

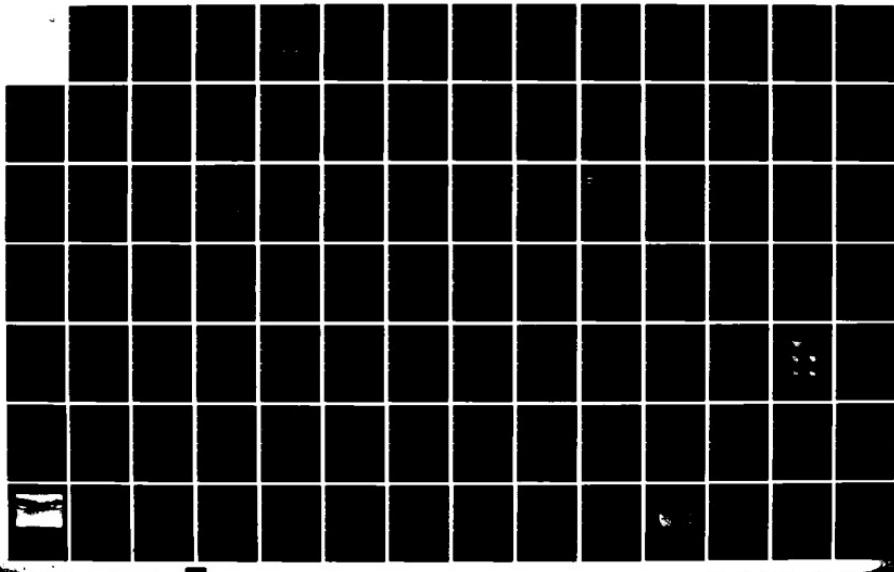
AD-A141 883 MULTIPROCESSOR Z-BUFFER ARCHITECTURE FOR HIGH-SPEED
HIGH COMPLEXITY COMPUTER IMAGE GENERATION(U) BOEING
AEROSPACE CO SEATTLE WA DEC 83 MDA903-82-C-0101

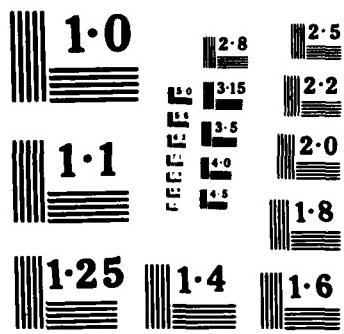
24

UNCLASSIFIED

F/G 9/2

NL





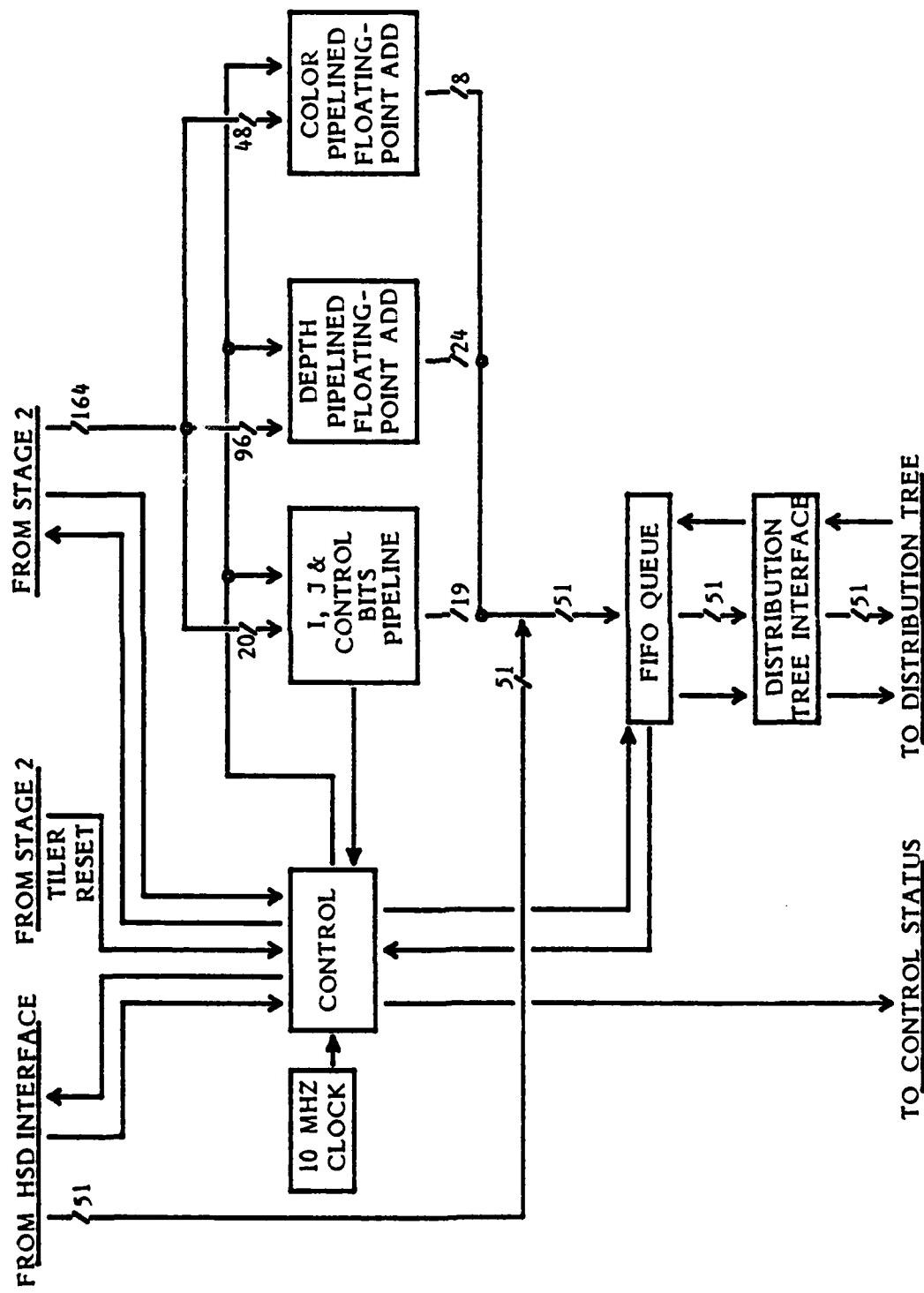
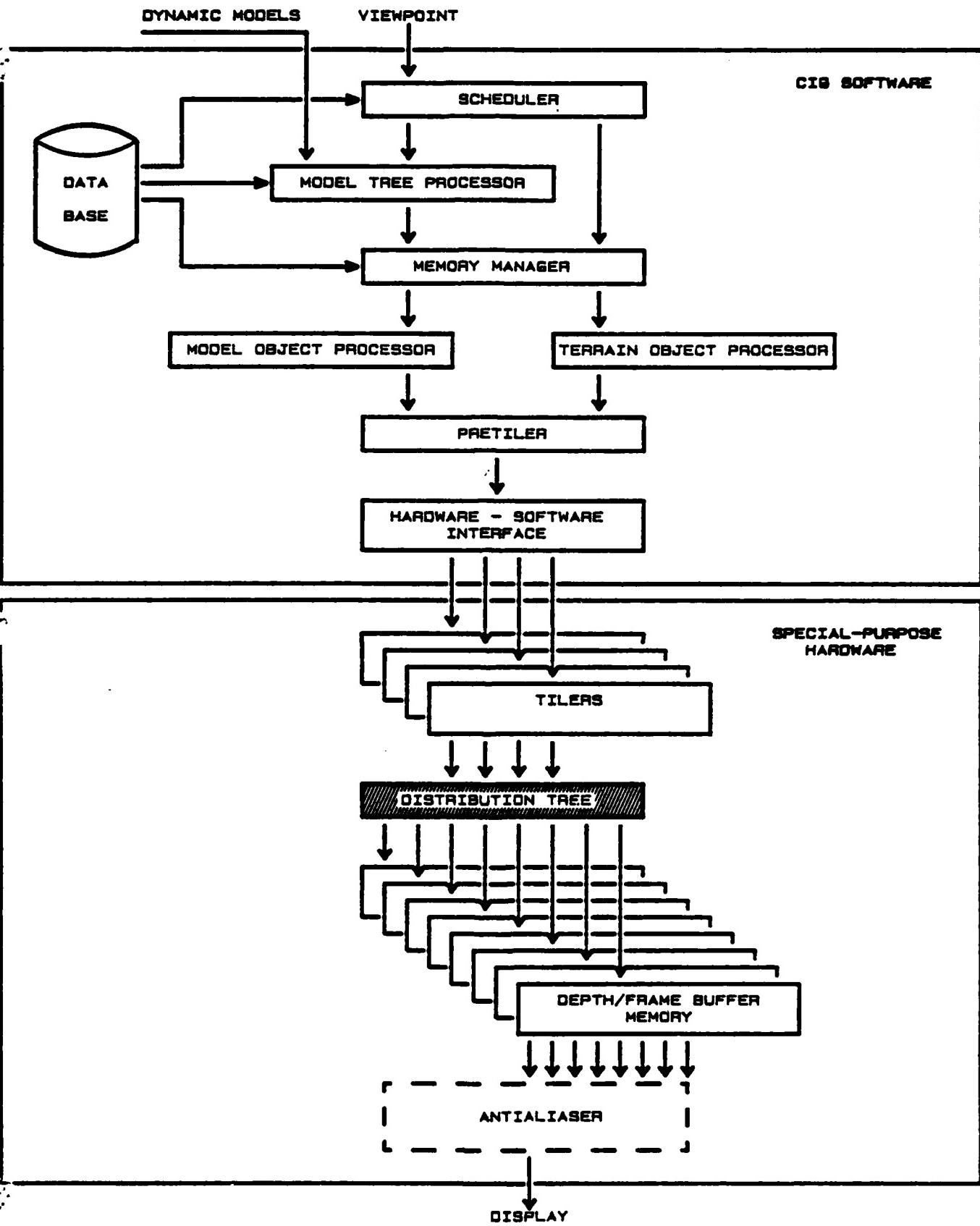


Figure 25. Tile Accumulate

4.2.5 FUTURE ENHANCEMENTS

Several enhancements would be implemented in future designs. Recent advances in high-speed arithmetic units and the results of a study on high-speed division algorithms would allow the correct color calculation to be implemented, as discussed in subsection 4.2.2.7. Full color (red, green, and blue) could be calculated either by interpolating each of the three primaries or by interpolating a single intensity and multiplying the result by a base color. Converting many of the internal data formats to floating-point would allow screen clipping to be performed in the tiling machine setup and would readily accommodate increases in screen resolution. The overall architecture of the tiler, however, has proven to be highly satisfactory and is not likely to change.



4.3 DISTRIBUTION TREE

4.3.1 INTRODUCTION

This section provides a survey of the multiprocessor-memory interconnection network problem.

A real-time computer image generation system requires an enormous amount of floating point computations to generate a high-complexity image. Even after using recursive algorithms to reduce the number of floating point computations, it still exceeds the capacity of a uniprocessor system.

Hence, multiple tilers and multiple memory modules operating in parallel are used in the CIG system.

To allow full utilization of each tiler and memory module, each tiler must be able to send data to any memory module. This interconnection problem is similar to multiprocessor-multiprocessor or multiprocessor-multimemory interconnection problems being considered in new supercomputer architectures. The network options that have been considered for these systems are described below.

4.3.2 INTERCONNECTION NETWORKS

Three types of interconnection networks were investigated: multiple-bus, crossbar, and multistage. A summary of cost/performance of interprocessor interconnection networks can be found in Reed and Schwetman (Reed83).

4.3.2.1 MULTIPLE-BUS

A single-bus system is based on a bus shared by all processors and all memory modules. Simultaneous accesses to the shared bus create conflict which is resolved by a network protocol. Commonly used protocols are time-division multiplexing access (TDMA) without priority, TDMA with a priority scheme, and carrier sensing multiple access (CSMA). If the bandwidth of the shared bus is much higher than the system data traffic rate, a single-bus system does provide a simple way of connecting multiprocessors to multiple memories. For a low utilization system,

chapter five of Kleinrock (Klei76) provides good insight into the analysis of a single-bus system.

However, as the number of processors increases or the system traffic rate increases, the system becomes limited by the bandwidth of the bus. In our system, each of the four tilers can simultaneously generate approximately 500 Mbits/sec. This means a shared bus has to sustain 2 Gbits/sec. data traffic, which is beyond current digital-bus transmission technology.

Marsan et al. (Mars83a, Mars83b) proposed several multiple-bus multiprocessor architectures using buses to allow processors to share memory modules. Their study showed that these architectures worked well for systems that managed bus contention by maximizing local memory accesses and minimizing bus utilization. However, these architectures were not a good match for the ARS. The requirement that each tiler be able to address any memory would increase bus utilization and contention. This creates a high-volume environment unsuited to the multiple-bus architectures.

4.3.2.2 CROSSBAR

A crossbar is a grid of switches that can be configured to directly connect any of N row elements to any of M column elements. As a processor-memory interconnection network, a crossbar would allow N processors to directly access M memories via NxM switches. The crossbar network is a valuable model for the analysis of processor-memory interconnection networks. Early works using the crossbar construct in the analysis of memory interferences can be found in references (Chan77, Bask76, and Bhan75). However, the crossbar network has some serious liabilities as an implementation strategy. The complexity of a crossbar network, measured by the number of switches, varies with the square of the number of processors, making the cost of implementation prohibitive for large numbers of processors. The hardware construct required to resolve contention when multiple processors request simultaneous access to the same memory module is not trivial. Also, a study done by Lang et al. (Lang82) indicated that a crossbar system has no better throughput than a multiple-bus system.

4.3.2.3 MULTISTAGE

Multistage networks are networks that are arranged as ordered levels of elementary switches. These switches essentially receive data and determine a routing path for that data. This network concept has been utilized for multiprocessor interconnections (Prem80) and multiprocessor-memory interconnections (Gott83, Barn83), and implemented in both packet-switching (Gott83) and circuit-switching (Barn83) environments.

Multistage networks can be designed with global data control determined by the network or local data control determined individually by each switch. In a circuit-switching or global data control environment, the switches receiving data are preset by a centralized control word. The data then takes a predetermined path through the network. In a packet-switching or local data control environment, switches are set by a destination tag that accompanies the data through the network. When data enters a switch, that switch interprets the destination tag and makes a routing decision based on that information.

As switches in the network are shared among different data paths, conflicts arise as multiple data streams destined for the same output of a switch arrive at the inputs of the switch simultaneously. These conflicts are called contention.

In a circuit-switching environment, contention is resolved by blocking the accesses of some memory requests. In a packet-switching environment, the contention is alleviated by queues internal to each switch.

Multistage networks constructed of 2x2 switches have been reported by many names: data manipulator (Feng73), baseline (Wu80a, Wu80c), banyan (Prem80), omega (Lawr75), flip (Batic76), delta (Pate81), and binary n-cube (Peas77). Wu and Feng (Wu80a) showed that these networks were topologically equivalent. They (Wu80b) also showed that admissible network functions of any of these networks could be obtained from the others by undergoing bit-reversal and/or bit-switching permutations.

The complexity of the multistage network grows on the order of $N_p \log_2 N_m$, which is significantly better than the complexity of the crossbar network at $N_p \times N_m$.

Also, the multistage network is known as a high-throughput architecture. Simulation results by Dias and Jump (Dias81) on a $(2^3 \times 2^3) = (8 \times 8)$ delta network indicated that, with sufficient buffering, the normalized throughput could be 97%. This indicates that data traffic is unaffected by contention at a maximum of 97% of the time.

The distribution tree construct used in the ARS hardware to connect the tilers to the memory modules was envisioned as the merging of four parallel sorting trees. It is a special form of a delta network described by Patel (Patel81). The distribution tree transfers data as packets, each containing its memory address. Each switch in the network uses the address to properly route the data and has internal buffers to store the data in case of contention. This type of multistage network provides the necessary high transfer rates that multibus and crossbar networks cannot.

4.3.3 ARCHITECTURE DEVELOPMENT

The modular design and performance of a multistage network make it an optimal choice for the tiler-memory interconnection network for this project. This section expands on that concept.

4.3.3.1 TILER-MEMORY CONFIGURATION

A stated objective of this project was to investigate high-speed multiprocessor architectures. To accomplish this, it was determined that a custom hardware system comprised of N_p processors (tilers) connected to N_m memory modules via a multistage interconnection network would be built. Choosing the number of tilers involved several tradeoffs. The combined polygon processing power of the N_p tilers would have to be approximately equal to that required by a real-time CIG system. The network would have to be sufficiently complex to allow its performance to be evaluated in a multiprocessor environment. Building a system that was too large would have unnecessarily burdened the design effort and costs. Therefore, it was decided that a four-tiler system would be adequate to achieve the objectives of this project.

Early evaluation of the memory module indicated that a memory access rate equal to the tiler output rate of 10 million pixels per second was unfeasible. The average access rate was estimated to be 6.67 million pixels per second with a worst case of 5 million pixels/second. Since the cumulative output rate of four tilers is 40 million pixels/second, the system would require eight memory modules to accommodate that rate. The resulting system is a $N_p \times N_m = N_p \times 2N_p$ interconnection network with $N_p=4$ tilers and $N_m=8$ memory modules.

4.3.3.2 2x2 SWITCH ELEMENTS

The design of the network should be expandable to support an increase in the number of tilers and memory modules. This can be accomplished by a modular design in which the network is built by interconnecting basic cells or switch elements. In a packet-switching network, each cell exercises local control over incoming data. Control within each cell depends only on status information from the cell itself and its interconnected neighbors. This approach eliminates the communication paths required by a circuit-switching network that connect a central controller to every cell in the network. A packet-switching network utilizes queues within the switch elements to resolve contention locally. A circuit-switching network handles contention globally by blocking accesses to the network. This global strategy causes processor inefficiency when contention rates are high by restricting input to the network. Therefore, switch elements designed to implement a packet-switching network were chosen for the ARS hardware.

The concept of a network that accepts input and routes this input through a series of switching elements to multiple destinations can be simplified into a general sorting problem. For example, sorting pixels from a tiler to eight memory modules is analogous to sorting a stack of cards representing eight different colors into eight stacks of the same color. Each cell of a 1x8 sorting machine could sort two primitive colors represented by '0' and '1'. Thus, eight colors could be represented by concatenating three primitive colors (C_2, C_1, C_0) - '000', '001', '010', '011', '100', '101', '110', and '111'. The 1x8 sorting machine shown in Figure 26 is made of seven cells, $S_{00} \dots S_{23}$. The first level of cells sorts cards according to the least significant bit (C_0) of the color. The second level sorts according to the next least significant bit (C_1), and the third level sorts on the most significant bit (C_2).

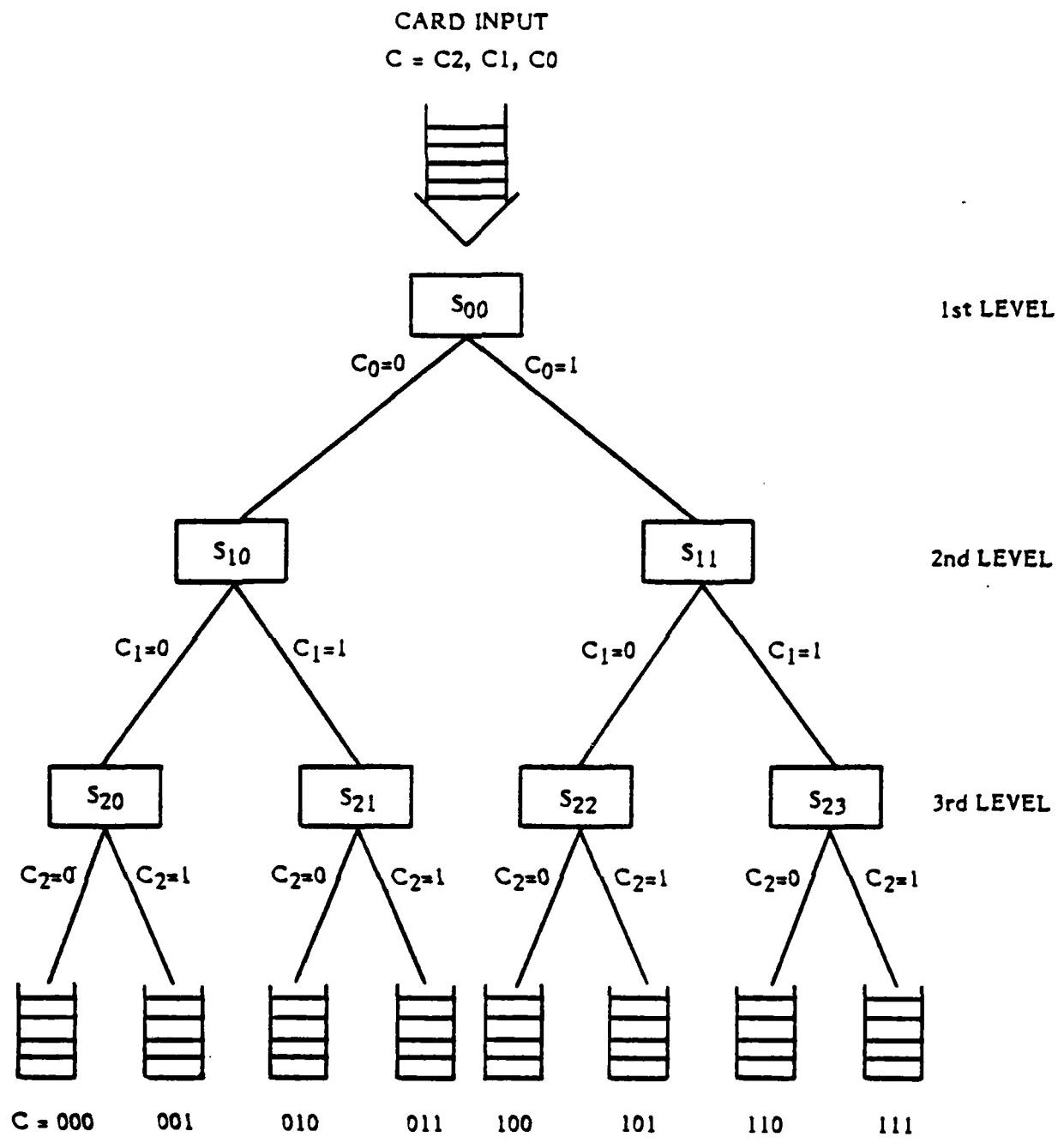


Figure 26. A 1×8 Sorting Machine

By making minor modifications to some of the 1x8 cells, a 2x8 sorting machine could be constructed that sorts two stacks of cards simultaneously. Figure 27 shows a 2x8 sorting machine, built by adding cell S_{01} to input the second stack and by modifying second-level cells S_{10} and S_{11} to accept the outputs of first-level cells S_{00} and S_{01} . Cells S_{10} and S_{11} now have two input and two output paths and are often referred to as 2x2 switches or 2x2 butterflies. When cells S_{00} and S_{01} both send cards '000' or '001' to cell S_{10} simultaneously, a conflict occurs. To resolve this conflict, cell S_{10} has to sort twice as fast, or store one of the cards and output it at some later time. This conflict is called contention and will be analyzed in subsection 4.3.3.3.

By merging two 2x8 sorting machines and modifying cells S_{20} , S_{21} , S_{22} , and S_{23} , a 4x8 sorting machine is developed as shown in Figure 28. This configuration matches the requirements of the architecture research system for a four-tiler, eight-memory interconnection network. This network could also have been derived from the reversed baseline network by disabling one input in each of the first level 2x2 switches.

The control of this 4x8 network or distribution tree is by a destination tag which is assigned to each pixel as it exits the tiler. This tag accompanies the data as it is routed through the tree. Each switch bases its switching decision on part of the information contained in this tag and status information supplied by its four connecting switches.

Since this type of distribution tree is derived from the basic 2x2 switch element, a tree of any size ($N_p \times N_m$) could be constructed. It would be modular in construction and its performance would be invariant to the network size, assuming a uniform memory access pattern.

4.3.3.3 CONTENTION ANALYSIS

In the preceding section, a modular distribution tree meeting the requirement for a four-tiler eight-memory interconnect network was described. The basic elements of the tree are a 1x2 switch and a 2x2 switch. The 2x2 switch is designed with two input ports and two output ports. When the switch simultaneously receives two inputs that are destined for the same output port, then the problem of data conflict or contention arises.

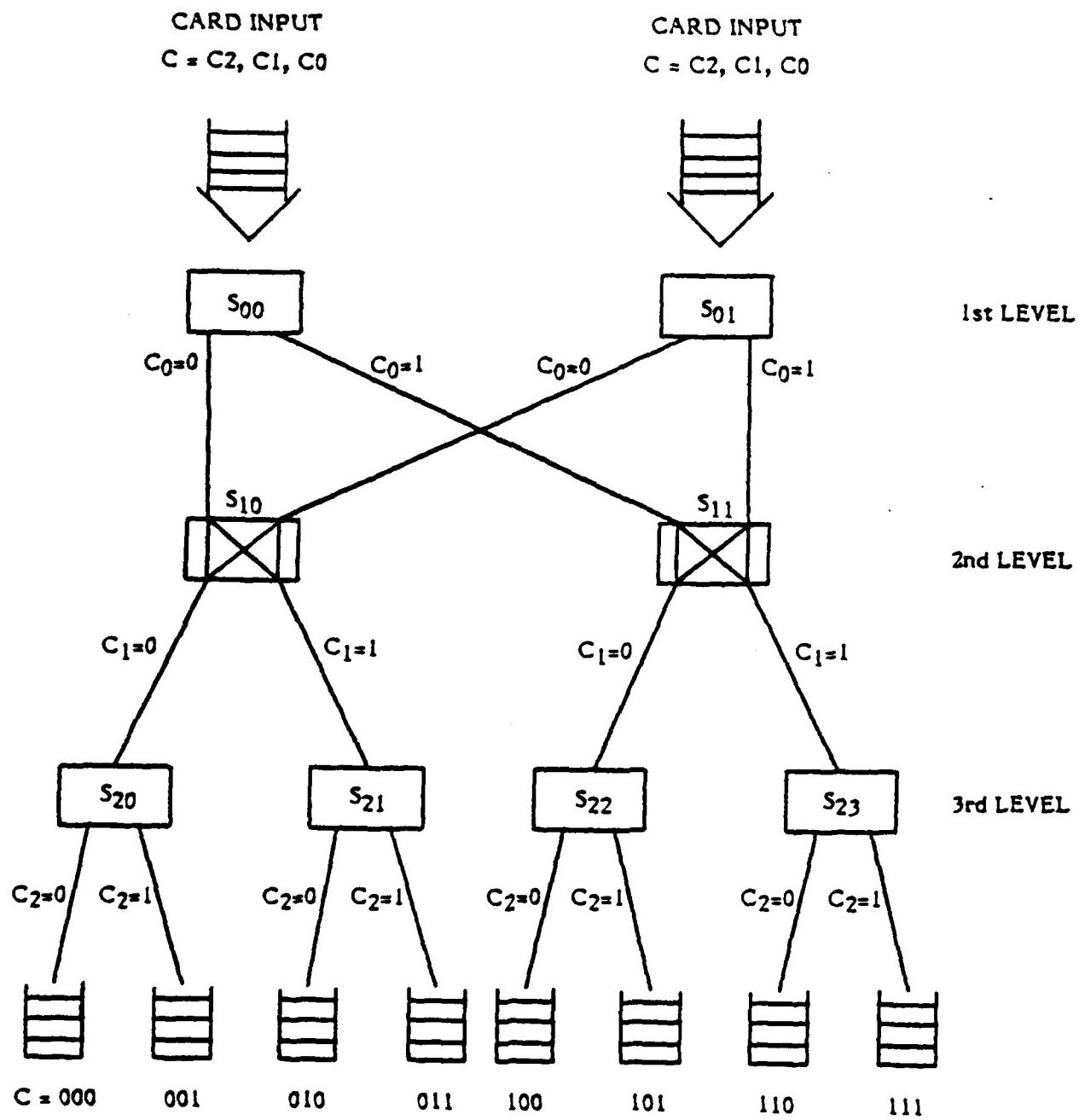


Figure 27. A 2×8 Sorting Machine

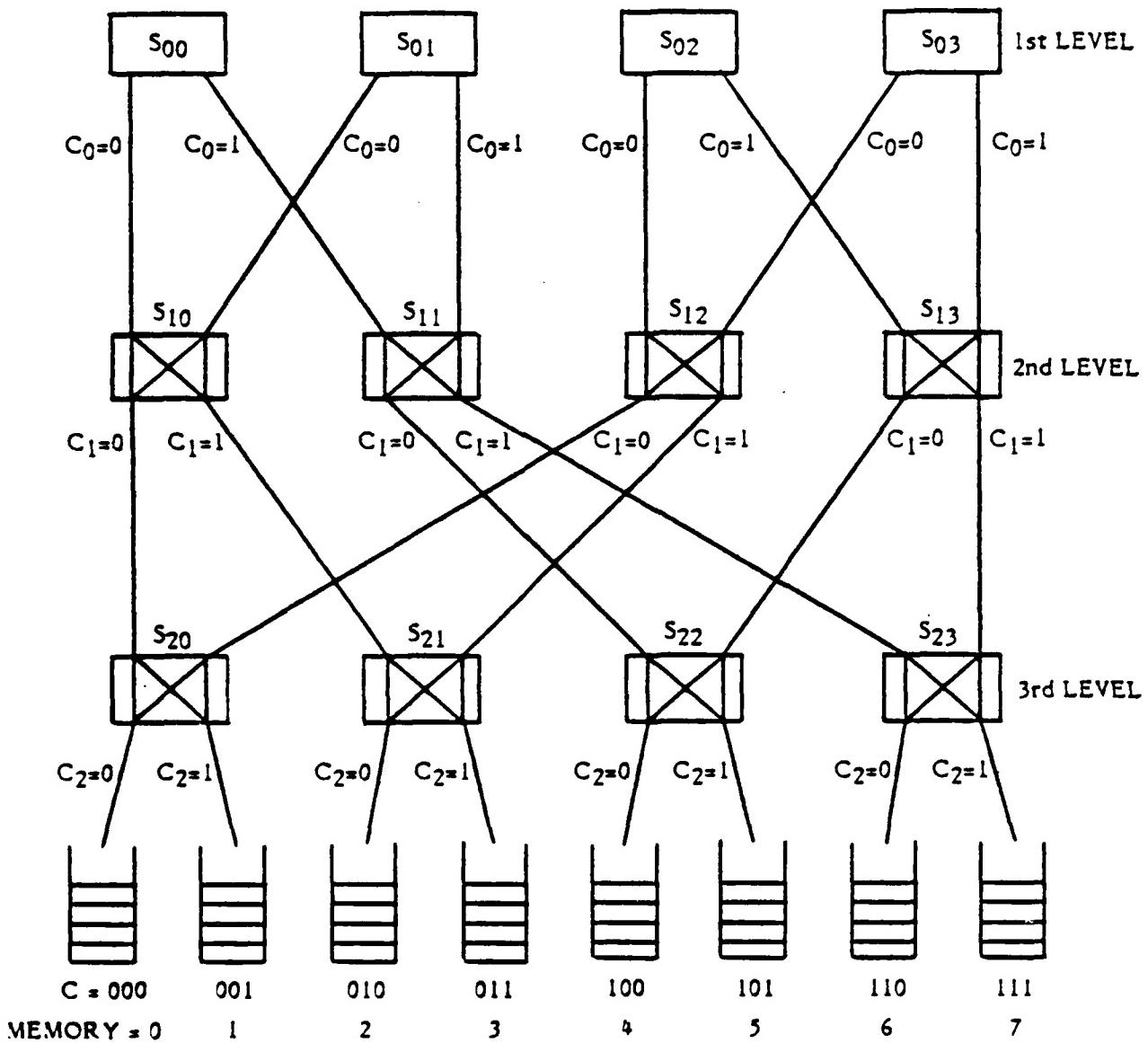
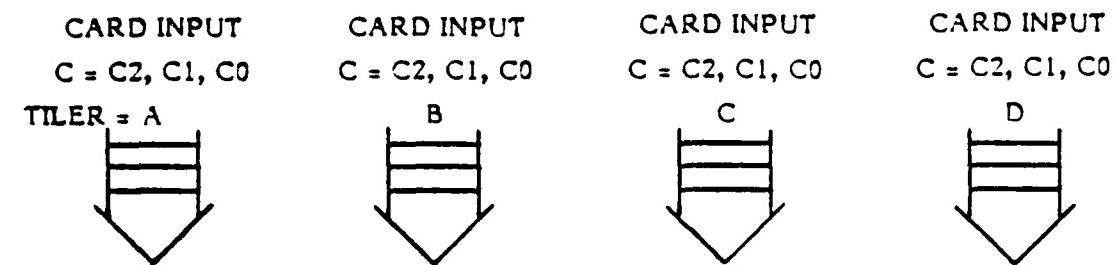


Figure 28. A 4x8 Sorting Machine

To resolve this contention, each level of 2x2 switches would have to operate at twice the speed of the previous level or have some form of storage to temporarily delay an output value. Doubling the speed at each level is unacceptable. The network would cease to be modular as the speed requirements for the lower levels and the memories became unrealistic. However, the idea of temporarily delaying output is a viable alternative. This concept is called queuing.

To resolve contention, one queue per output port should be available in each switch to store conflicting pixel values for that port. The number of storage levels required in each queue is dependent on the data traffic through the tree. Therefore, minimal conflict resulting in low queue utilization and minimum storage requirements is desirable.

One approach to minimizing contention is distributing data traffic uniformly over the entire network. Consider the following example, referring to Figure 28. Tiler A inputs into switch S_{00} a contiguous block of pixels destined for memory 0 and tiler D simultaneously inputs into switch S_{03} a contiguous block of pixels also destined for memory 0. Obviously, the data traffic is not uniform over the entire network and the contention at switch S_{20} is considerable. To distribute the data traffic more evenly, a hashing function could be implemented. This hashing function would alter the destination tag of adjacent pixels so that they route to a different memory.

Using the vertical (i) and horizontal (j) screen coordinates as the destination tag of a pixel, then an expression describing a hashing function (Bui75b) is:

$$MA = (3j + i) \text{ MOD } (8)$$

where MA = memory address = altered destination tag

The net effect of this hashing function or scrambler is that adjacent pixels are placed 3 memories apart. For the previous example, the hashing function would alter the destination tag as follows.

Pixel Number	1	2	3	4	5	6	7	8	9	...
i	0	0	0	0	0	0	0	0	0	...
j	0	1	2	3	4	5	6	7	8	...
Memory Address	0	3	6	1	4	7	2	5	0	...

This hashing function will clearly route adjacent pixels along different paths, thereby evening out the data traffic and reducing contention.

An important consideration in the design of a 2x2 switch is the length of its internal queues. In the ARS, the tilers and the network process pixels at the same rate. The input rate of the memories is only 75% of the tiler-network rate. If the data output by the tilers is evenly distributed among the memories, the A-cell switches split the input data equally between its two output ports. This, in effect, halves the data rate of each output port. The network of B-cells then evenly distributes the data so that any cell in the network experiences the same data rate. Since the data rate is halved by the A-cells, the input rate to any B-cell is half its potential. M/M/1 queuing theory for such a system predicts:

$$\rho = \frac{\text{input rate}}{\text{potential output rate}} = \frac{50}{100} = .5$$

The potential output rate of the B-cells that connect to the memories is reduced 75% by the slower speed of the memories. For these switches M/M/1 theory predicts:

$$\rho = 50/75 = 2/3$$

$$\text{average queue length} = \frac{\rho}{1-\rho} = 2$$

On the average, the tilers do output evenly distributed data. However, over the short term, the data may be biased toward one memory. These short-term biases cause increases in the switch queue lengths. The worst case for queue lengths is

when all tiler outputs are biased toward the same memory. To estimate the effect of such short-term biases on the switch queue lengths, a Gaussian distribution of output memory addresses centered around memory j was assumed for each tiler for 100 cycles. The transfer of this data through the network was simulated for the duration of the 100 cycles, and the average queue sizes were measured. Figure 29 shows the worst average queue lengths at each network level for networks of various sizes. The time required to clear these backed-up queues decreases the efficiency of the network. Figure 30 shows this decreased efficiency as a decrease in the effective number of processors. The results of the simulation demonstrate that, with a minimal amount of queuing, the distribution tree can sustain short-term biases in the output distributions and maintain an effective number of processors.

4.3.4 HARDWARE DESIGN OBJECTIVES

The primary design objectives were to implement an interconnect network fast enough to support the output data rate of the tilers and flexible enough to permit analysis of network characteristics.

4.3.4.1 OPERATIONAL MODES

In the previous subsection, the concept of a hashing function was developed to evenly distribute data traffic through the network. To support evaluation of this concept, three modes of network operation were implemented. The first mode is referred to as the sequential mode and allows each pixel to be routed according to its destination tag. The second mode is an interleave mode, routing adjacent pixels to adjacent memories. The third mode is the scrambled mode which was discussed in subsection 4.3.3.3.

4.3.4.2 QUEUING

It was previously determined that each 2×2 switch in the distribution tree would have a queue for each output port. Figure 29 shows that the worst-case average queue length required by an eight memory system is about sixteen levels. However, the ARS queues are 256 levels deep. From a hardware design standpoint,

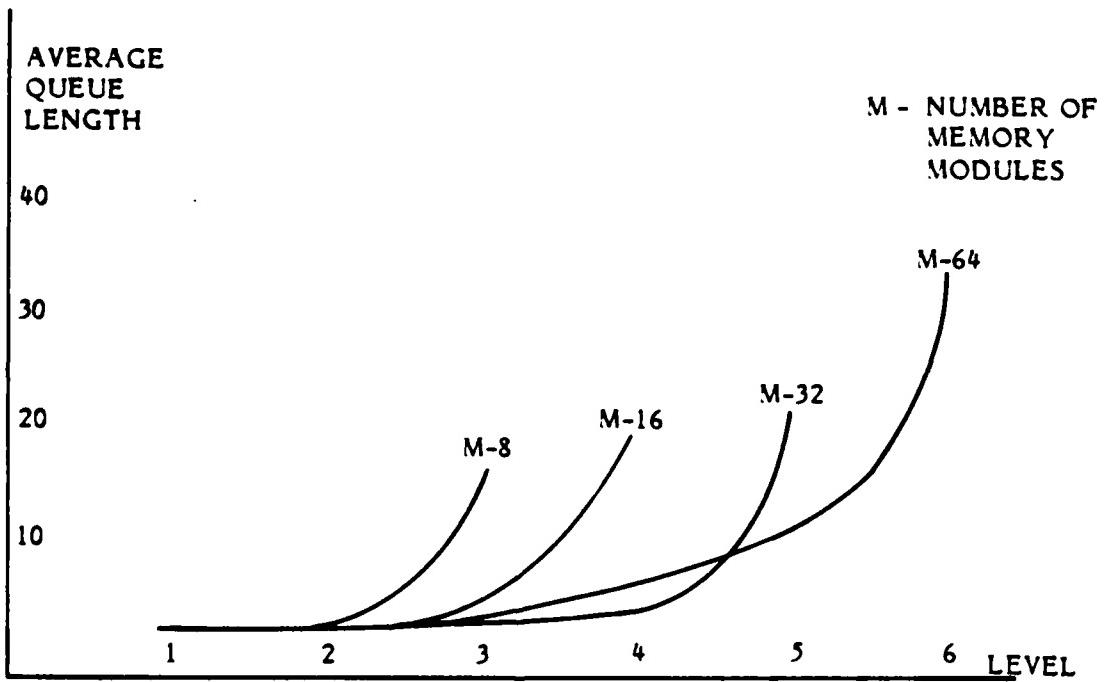


Figure 29. Cell Queue Length ~ Tree Level

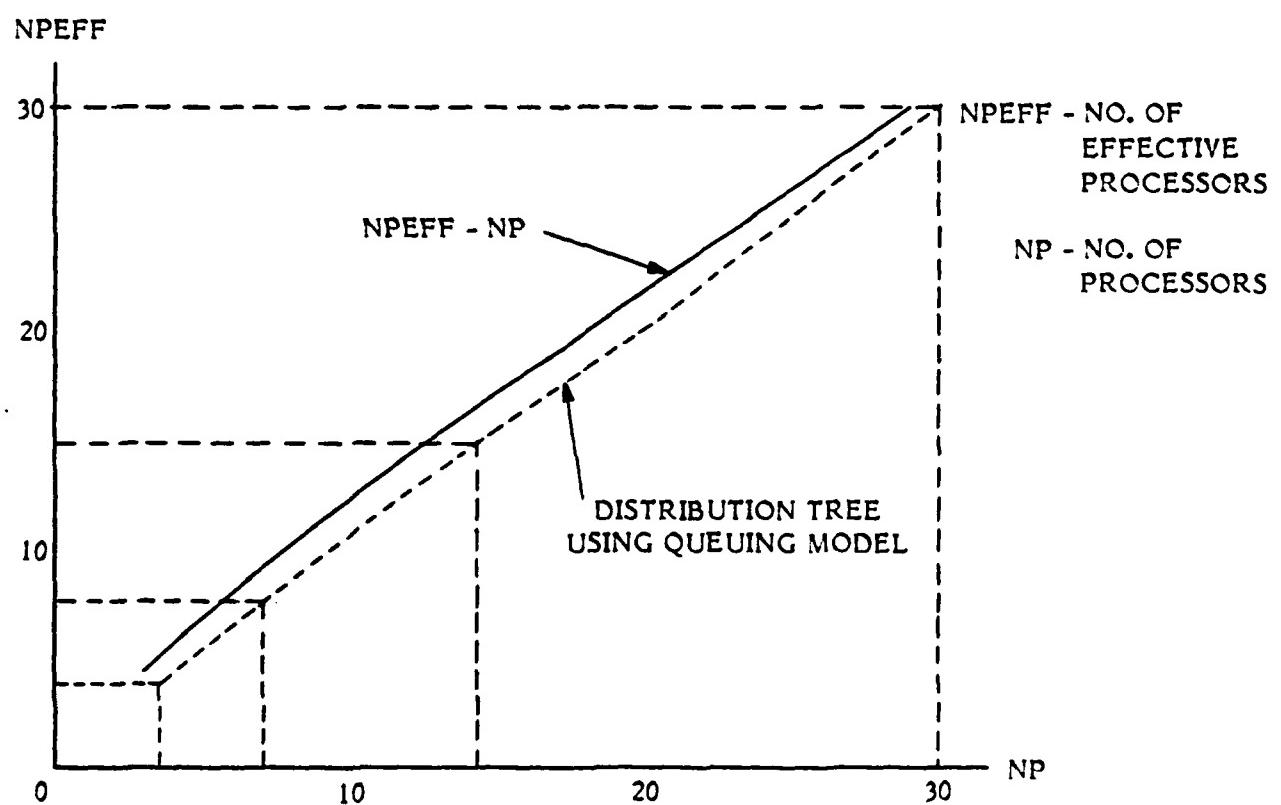


Figure 30. Tree Contention by Queuing Model

implementing a queue of 256 levels has no greater system impact than implementing a queue sixteen levels deep because there is no difference in component count or circuit complexity.

4.3.4.3 ELECTRICAL AND MECHANICAL

As with all the circuit cards fabricated for the ARS, there were few constraints on the electrical and mechanical characteristics of the distribution tree. The tree would be fabricated on four wire-wrap circuit cards and consume no more than 400 watts of 5 VDC power. The hardware design would consist of two sets each of two different card level designs. One design implements two upper level 1x2 switches and two second level 2x2 switches. The second design implements two 2x2 switches and the corresponding tree-memory interface circuits.

4.3.5 HARDWARE DESIGN OVERVIEW

The distribution tree is a modular design whereby cells of a standard configuration are interconnected to form a network. This section will provide an overview of the tiler-tree and tree-memory interface designs, and the 1x2 and 2x2 switch designs.

4.3.5.1 INTERFACES

The input path of each 1x2 switch or A-cell connects directly to a tiler. The tiler-tree interface (see Figure 31) is a simple data-available, data-accepted handshake that supports a 10 million pixel/second data rate. The pixel data path is 51 bits wide and consists of an 18-bit memory address or destination tag, 24-bit depth value, 8-bit color value, and a 1-bit background flag. The tree-memory interface circuitry connects the output ports of the lowest level 2x2 switches to the memory input ports. First-In/First-Out (FIFO) queues that are 64 levels deep are used to buffer each switch output port to compensate for the difference in memory and tree cycle times. The tree-memory interface is also a simple data-available, data-accepted handshake designed to support a 10 million pixel/second data rate.

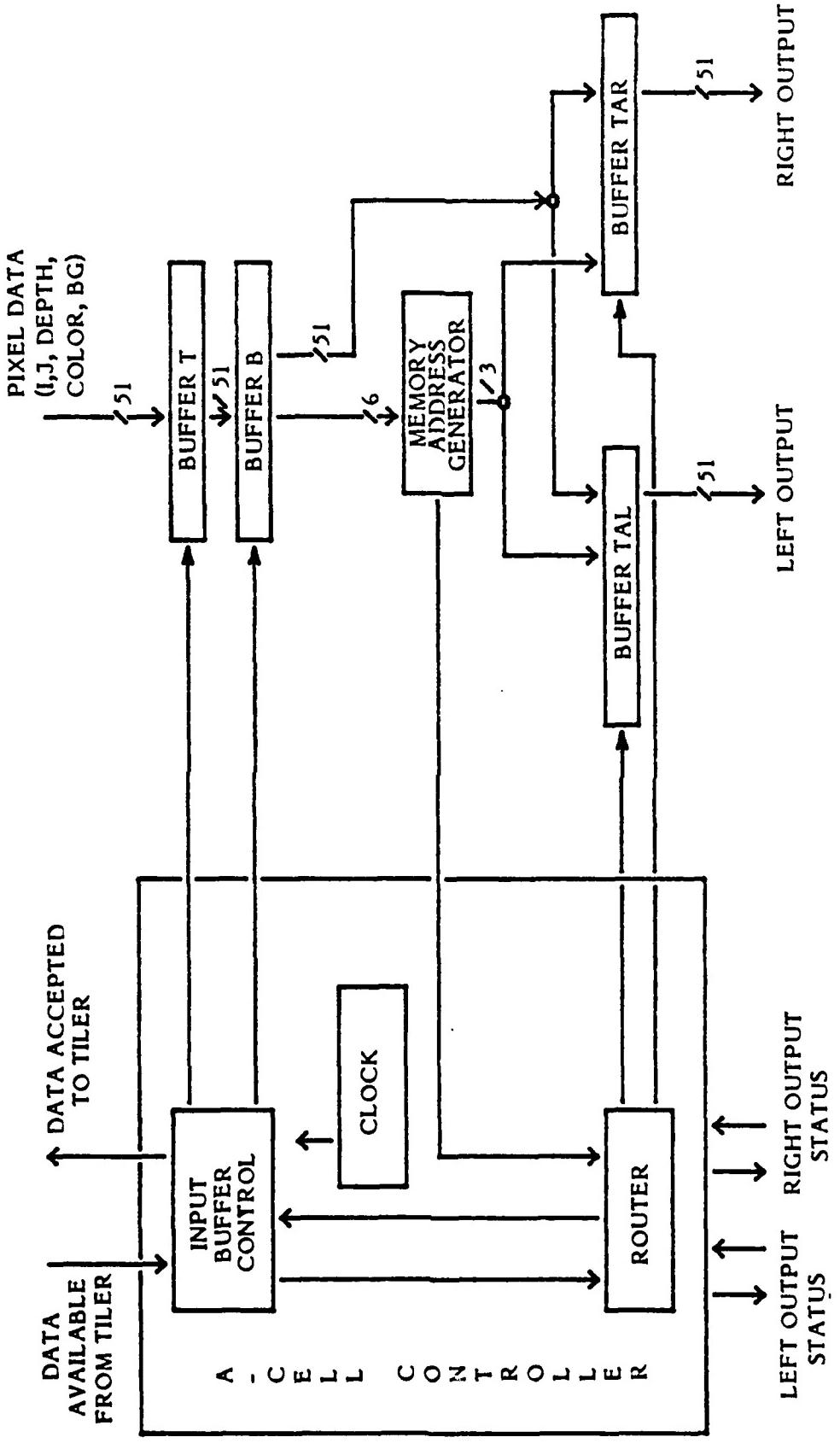


Figure 31. Type A-cell Functional Diagram

4.3.5.2 A-CELL

The A-cell block diagram shown in Figure 31 demonstrates the simplicity of the 1x2 switch. The A-cell controller directs the data traffic through storage buffers T, B, TAL, and TAR. The decision to route a pixel to the right output port or left output port is based on the most significant bit of the destination tag. Status information is passed between the A-cell controller and the 2x2 switches (B-cells) connected to the A-cell outputs. This status information indicates that data is available at the A-cell outputs and that the data has been accepted by the 2x2 switches. Table 9 provides a summary of the A-cell controller algorithm. The memory address generator is a simple address look-up table that implements the three modes of operation previously discussed, e.g., sequential, interleave, and scramble.

4.3.5.3 B-CELL

The B-cell accepts data from an upper-level A-cell and stores it in the upper buffers (see Figure 32). Several factors can now influence how data is routed through the B-cell to the output ports. An input multiplexer is used to route data from the input buffers to the queues if the control logic indicates queuing is required due to contention at the output port. An output multiplexer routes data from the input buffers directly to the output buffers when queuing is not required.

The B-cell controller is implemented as a finite-state machine that cycles at 10 MHz. The controller receives certain information every cycle and uses this information to determine what action needs to be taken. This information includes the status of the input and output ports, queue condition (empty, not empty, full), and the destination tags of input data. The B-cell controller algorithm is rather lengthy, but the concept is represented by the example in Table 10. The routing bit refers to that bit of the destination tag tested by the controller to determine if the data is to be routed to the left output port (routing bit = 0) or to the right output port (routing bit = 1).

Table 9. A-cell Controller Algorithm

START CYCLE		
IF	(TAL data accepted)	AND
	(Buffer B is not empty)	AND
	(Routing bit = 0)	
THEN BEGIN		
	(Buffer B → Buffer TAL)	AND
	(Post TAL data available status)	
END		
IF	(TAR data accepted)	AND
	(Buffer B is not empty)	AND
	(Routing bit = 1)	
THEN BEGIN		
	(Buffer B → Buffer TAR)	AND
	(Post TAR data available status)	
END		
IF	(Buffer B → Buffer TAL)	OR
	(Buffer B → Buffer TAR)	
THEN	(Buffer B is empty)	
IF	(Buffer B is empty)	AND
	(Buffer T is not empty)	
THEN BEGIN		
	(Buffer T → Buffer B)	AND
	(Buffer B is not empty)	AND
	(Buffer T is empty)	
END		
IF	(Buffer T is empty)	AND
	(Tiler pixel data available)	
THEN BEGIN		
	(Tiler pixel data → Buffer T)	AND
	(Tiler pixel data accepted)	AND
	(Buffer T is not empty)	
END		
END CYCLE		

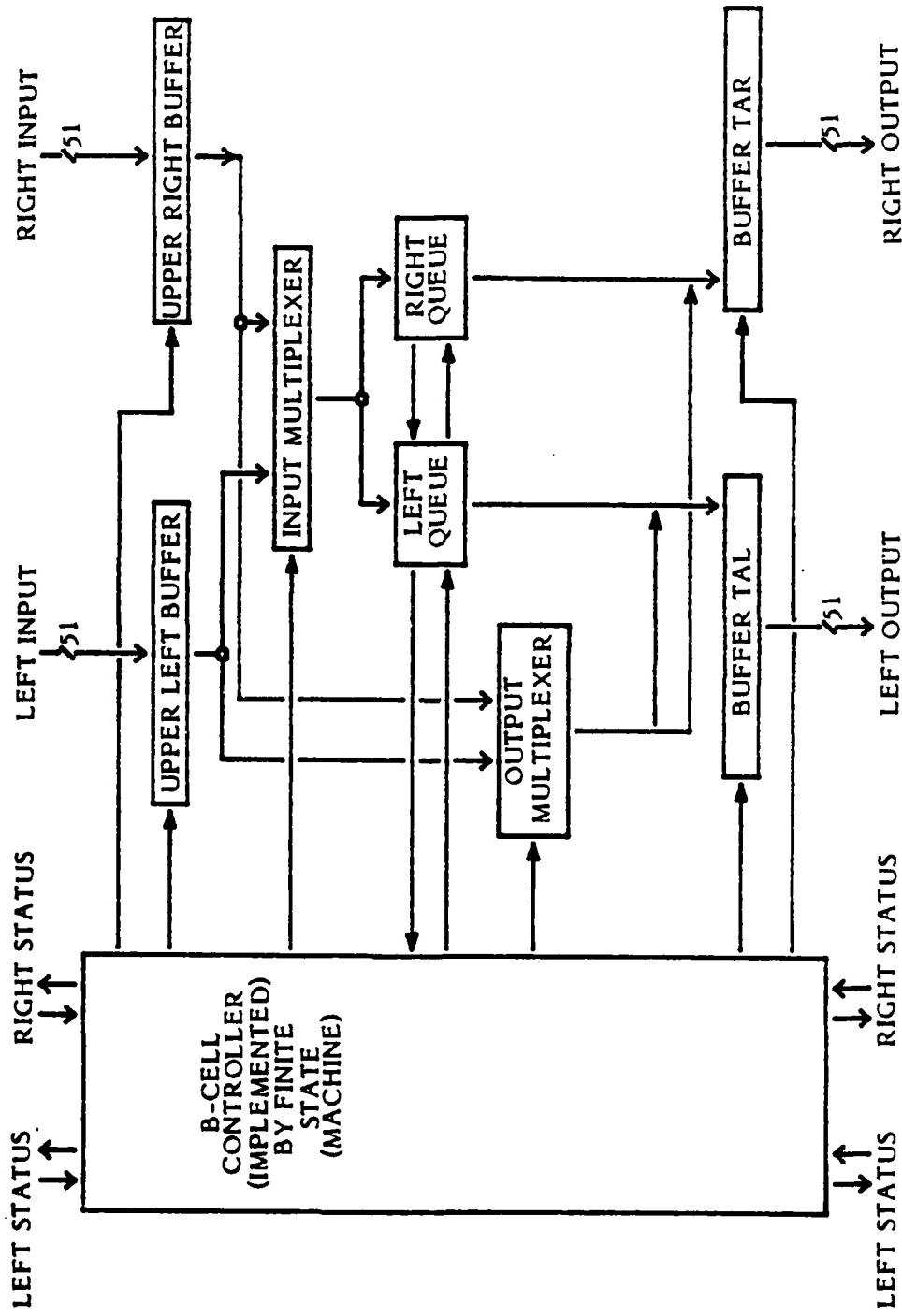


Figure 32. Type B-cell Functional Diagram

Table 10. B-cell Controller Algorithm
 (Continued on following page)

```

START CYCLE
IF          (Buffer TAL is empty)
THEN BEGIN
    IF      (Upper left buffer is not empty)      AND
           (Upper left routing bit = 0)
    THEN BEGIN
        (Upper left buffer → Buffer TAL)      AND
        (Upper left data accepted)            AND
        (Post TAL data available status)
    END
ELSE
    IF      (Upper right buffer is not empty)     AND
           (Upper right routing bit = 0)
    THEN BEGIN
        (Upper right buffer → Buffer TAL)      AND
        (Upper right data accepted)            AND
        (Post TAL data available status)
    END
ELSE
    IF (Left Queue is not empty)
    THEN BEGIN
        (Pop left queue → Buffer TAL)      AND
        (Post TAL data available status)
    END
ELSE
    IF      (Upper left buffer is not empty)      AND
           (Upper left routing bit = 0)          AND
           (Left queue is not full)
    THEN BEGIN
        (Upper left buffer → left queue)      AND
        (Upper left data accepted)
    END

```

Table 10. B-cell Controller Algorithm (Concluded)

```
ELSE      (Upper left data not accepted)
IF        (Upper right buffer is not empty)      AND
          (Upper right routing bit = 0)            AND
          (Left queue is not full)

THEN BEGIN
          (Upper right buffer → left queue)      AND
          (Upper right data accepted)

END
ELSE      (Upper right data not accepted)
END CYCLE
```

4.3.6 FUTURE DEVELOPMENTS

A four-tiler, eight-memory interconnection network that met project specifications was designed and implemented. However, several areas of the distribution tree design require additional research before incorporation into a real-time CIG system.

4.3.6.1 ARCHITECTURAL IMPROVEMENTS

The distribution tree is a high-performance packet-switching network designed to handle data traffic with a high probability of contention. The performance evaluation section (see Section 7.0) shows that actual contention rates on the distribution tree are low and internal queue lengths are minimal when the scrambled mode is selected and tiler efficiency is below 100%. In view of these low contention rates, other network architectures can be considered. One possible alternative is the circuit-switching multistage network previously discussed in subsection 4.3.2.3.

The circuit-switching network shares several advantages of the packet-switching network. It is a modular system, and network connection time will support the tiler output rate (Barn83).

Although contention is resolved in a different manner, conflicting accesses in a circuit-switching network can be cleared in a maximum of six network clock cycles (ParkD80). Implementing a circuit-switching network would reduce the complexity and size of each switch by removing the internal queues and associated control logic from the switch. For future network implementations, the circuit-switching network would have advantages if error correction is required. A packet-switching error detection and correction scheme requires that a multiple bit correction code be transmitted with each data word. The advantage of this approach is that data integrity could be verified and, if necessary, corrected locally at each level. Drawbacks to this scheme are a significant increase in word width and the addition of detection/correction circuitry. A circuit-switching approach would involve a simpler means of error detection such as parity checks. If bad data is detected, retransmission would be requested.

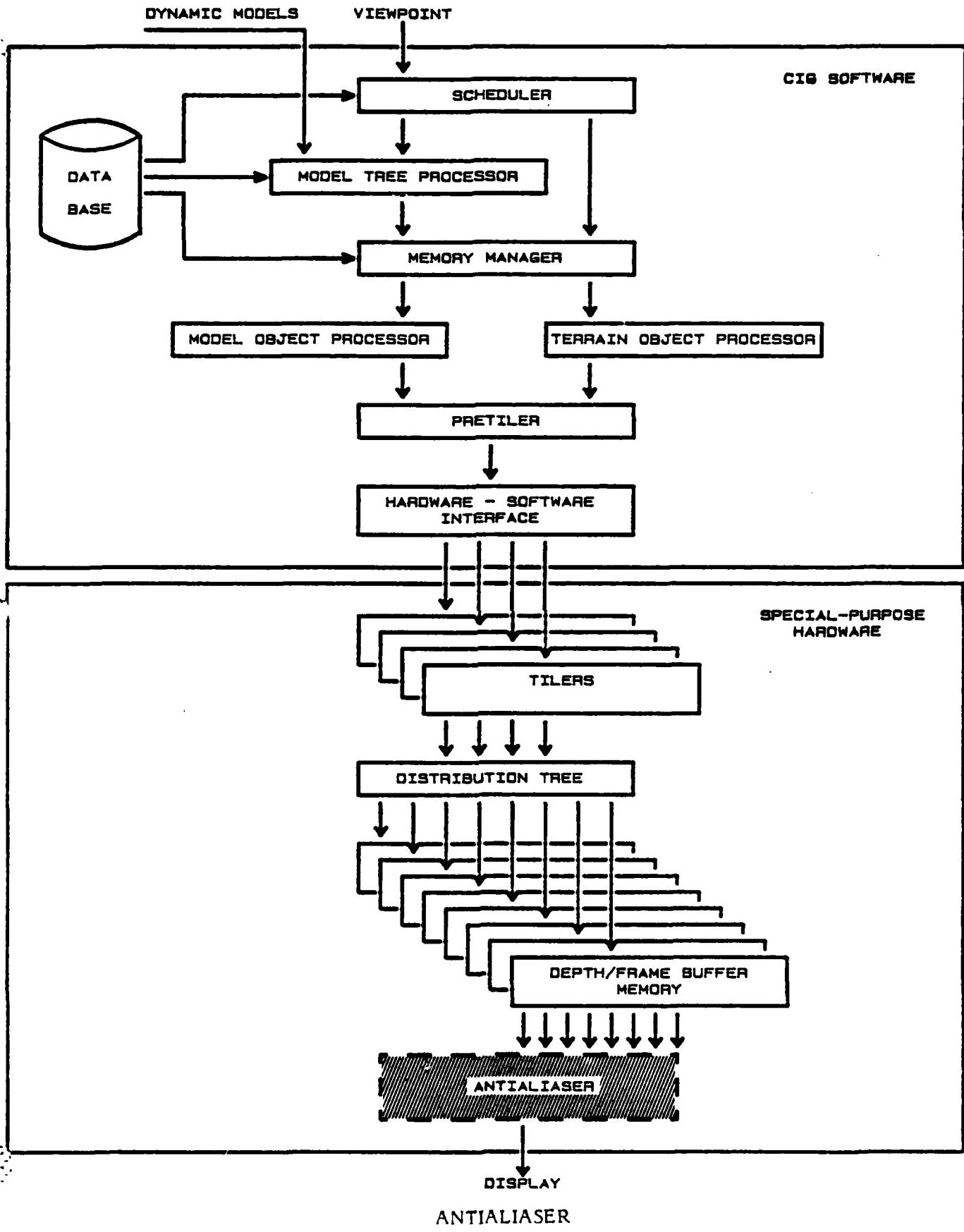
Additional study directed toward understanding the low contention rates in the ARS distribution tree and the impact data transmission errors have on system performance should provide insight into the feasibility of other architectures such as the circuit-switching approach.

4.3.6.2 IMPLEMENTATION IMPROVEMENTS

As the complexity of the network increases, managing the number of interconnections and the physical size of the network becomes a significant problem. The data word processed by the system could exceed 80 bits in future CIG systems. Transferring this data in parallel would result in a very large number of interconnections per card. For example, a 2x2 switch has four data paths, two input and two output. Implementing this switch with an 80-bit data word on a single circuit card would require a minimum of $4 \times 80 = 320$ interconnections. A similar 8x8 switch would require 1280 connections per card. The 14 inch by 17 inch cards currently in use provide only 564 off-board connections. Therefore, other data transfer methods, such as serial data transmission and partitioning of the data path, need to be investigated in order to reduce the number of interconnections per card.

The larger data word also creates increased pin counts internal to each switch. Pin count could be reduced by refining the switch design through incorporation of recently available integrated circuits. Another method of reducing the pin count and physical size would be implementation of the switch in a custom-design integrated circuit. A prototype 2x2 switch implemented with a 12-bit data path and 8-level deep queues was designed and fabricated. Characterization tests indicated that the device was fully functional up to 2MHz, although the design specification called for 10MHz operation to ensure compatibility with the distribution tree. Further analysis showed that a minor design change would correct the speed discrepancy. Implementing a network utilizing these devices would significantly reduce the pin count and physical size, but would not solve the interconnection problem.

A great deal of insight into the problem of interconnecting multiple processors and multiple memories has been gained through the construction and analysis of the ARS distribution tree. This basic architecture could be refined by additional research to provide an excellent choice for a high-performance interconnection network.



4.4 ANTIALIASING

4.4.1 ALIASING EFFECTS AND ANTIALIASING

Aliasing effects arise in raster graphics displays due to the fact that discrete sample points are displayed over finite areas on the screen (pixels) (see Figure 33). These effects are noticeable along high contrast edges in an image and occur as stair-stepping, crawling, line breakup and scintillation (see Figures 34-36).

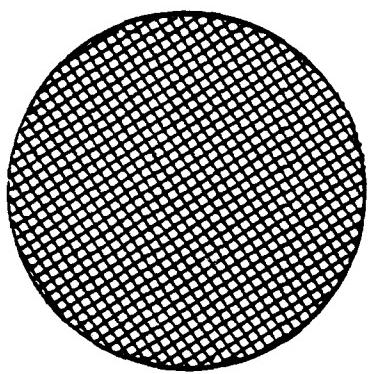
Antialiasing techniques increase the amount of edge information in the image in order to correctly blend colors over a pixel (Figure 37). The correct method of blending is to integrate the illumination function over the area covered by each visible face in a pixel (Crow77). Since this method is not practical for real-time image generation, some type of discrete approximation is required.

The current implementation of antialiasing oversamples the scene by a factor of four, and then computes the final image by forming a weighted-sum filter average of nine sample points (Figure 38). However, dynamic sequences antialiased in this manner still exhibit unsatisfactory aliasing effects. While oversampling increases memory requirements, further research has indicated that a higher amount of oversampling, together with proper filtering, can generate satisfactory images for real-time displays and still remain cost effective.

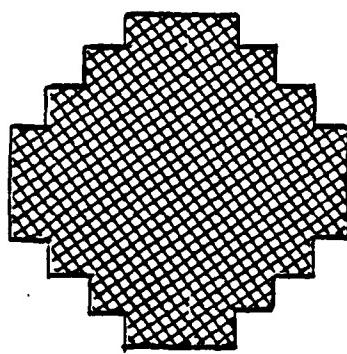
4.4.2 DESIGN OBJECTIVES

The antialiaser must be capable of accepting color data from the frame/depth buffer at 10 MHz. The input data is logically arranged as a 512x512 array of colors, and the antialiaser may read each color only once, in order, from left to right, and top to bottom.

The design consists of two levels. The first level filters in the horizontal direction, and the second level filters in the vertical direction. The first level reads rows of color data and filters the colors in overlapping groups of three (see Figure 39). Specifically, the first and third colors are added; the sum is right-shifted and added to the second color; and this sum is right-shifted and passed to the next level.

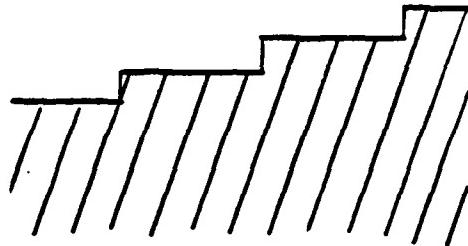


OBJECT TO BE DISPLAYED

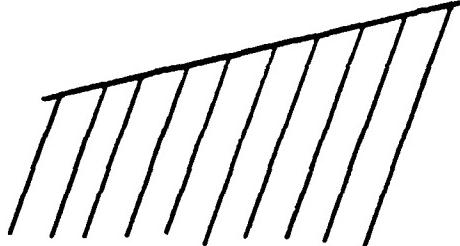


DISPLAYED OBJECT SHOWING
THE EFFECTS OF DISCRETE
SPATIAL SAMPLING

Figure 33. Aliasing Effects - Discrete Sampling



DISPLAYED EDGE SHOWING THE
EFFECTS OF STAIRSTEPPING



EDGE TO BE DISPLAYED

Figure 34. Aliasing Effects - Stairstepping

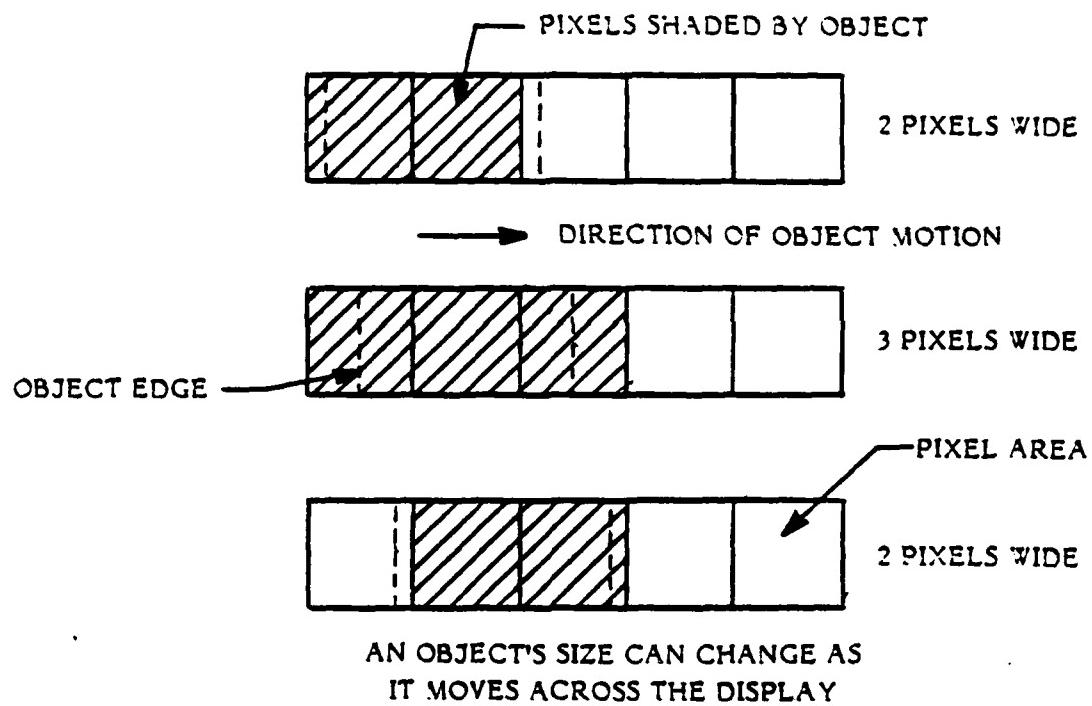


Figure 35. Aliasing Effects - Crawling

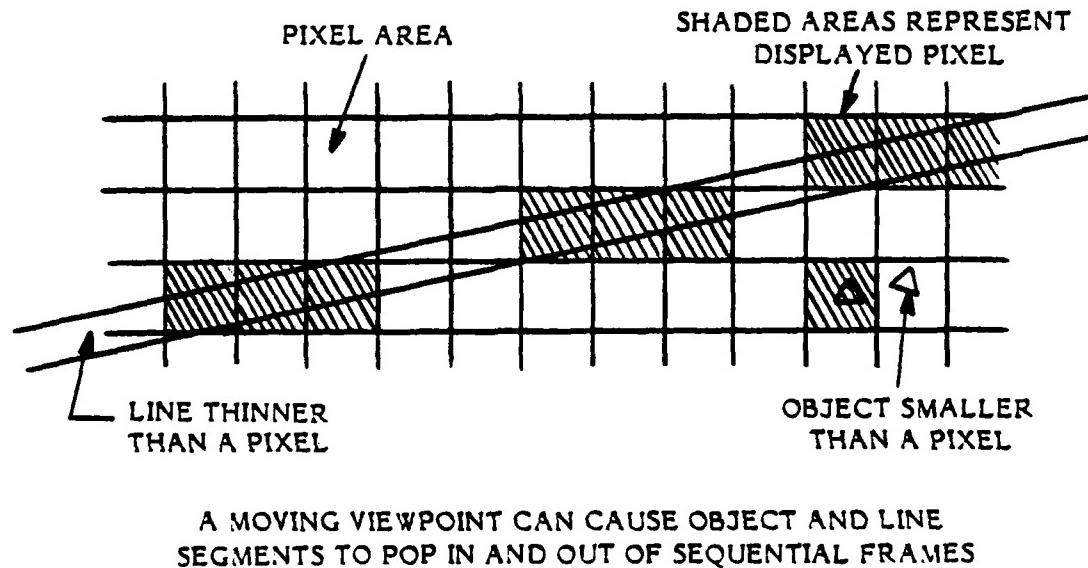
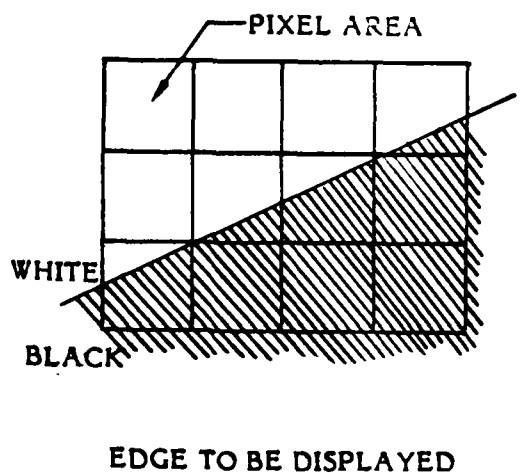


Figure 36. Aliasing Effects - Scintillation



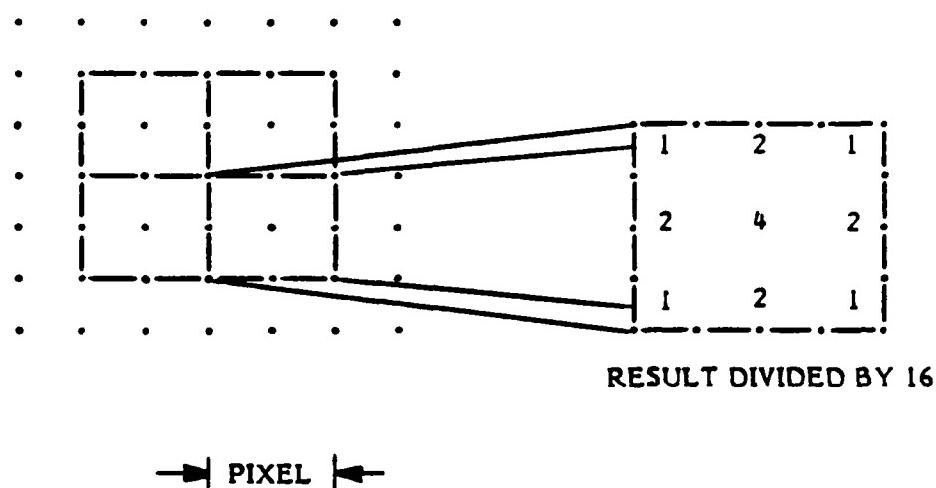
W	W	W	W
W	W	B	B
B	B	B	B

NO ANTIALIASING

W	W	W	WB
W	WB	BW	BW
BW	B	B	B

ANTIALIASING APPLIED

Figure 37.



EACH DISPLAY PIXEL IS THE WEIGHTED SUM AVERAGE OF 9 SAMPLE POINTS

Figure 38. 4x Oversampling and Filtering

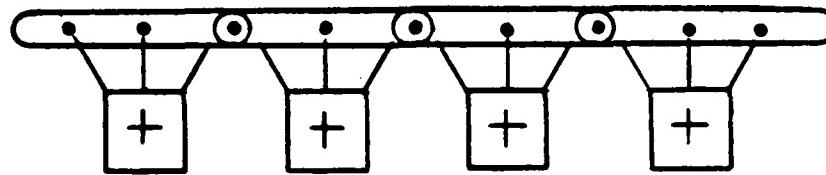


Figure 39. Level One Data Combination

The second level performs the same operations as the first, except that the three input colors are delayed relative to each other in order to combine pixels in the same column of successive rows. Thus, the second and third input colors are the outputs of a 256 and 512 stage delay register, respectively. The remainder of the second level is the same as the first level and results in the desired weighted sum.

The nine point filter used requires an input array of 513x513. Since only a 512x512 array is available, the antialiaser needs to fabricate an extra row and an extra column. The compromise chosen was to duplicate the top-most row and the left-most column, as shown in Figure 40.

C _{1,1}	C _{1,1}	C _{1,2}	C _{1,3}	·	·	·	C _{1,512}
C _{1,1}	C _{1,1}	C _{1,2}	C _{1,3}	·	·	·	C _{1,512}
C _{2,1}	C _{2,1}	·					
C _{3,1}	C _{3,1}						
C _{4,1}	C _{4,1}						
C _{5,1}	C _{5,1}						
·	·						
·	·						
·	·						
C _{512,1}	C _{512,1}						C _{512,512}

Row 0 is a duplicate of row 1 and
Column 0 is a duplicate of column 1

Figure 40. Logical Input to the Antialiaser

4.4.3 IMPLEMENTATION

The coefficients of the nine-point filter used were shown in Figure 38. This filter lends itself to simple implementation since the coefficients are multiples of two; the multiplication reduces to left shifts, and the final division by sixteen reduces to right shifts.

4.4.4 OPERATIONAL CHARACTERISTICS

The antialiaser which was designed is capable of processing input data at a rate of 10 MHz. Because the averaging results in a four-to-one reduction of data, the 10 MHz input rate results in an average output rate of two and a half megahertz.

The antialiaser latency is dictated by the time it takes to read in the data necessary for the first output. At first, this would appear to be two full rows and

three elements of the third row, totaling 1029 elements. However, since the topmost row is duplicated to form row zero, and the leftmost column is duplicated to form column zero, the total number of data inputs required before the first output is 514.

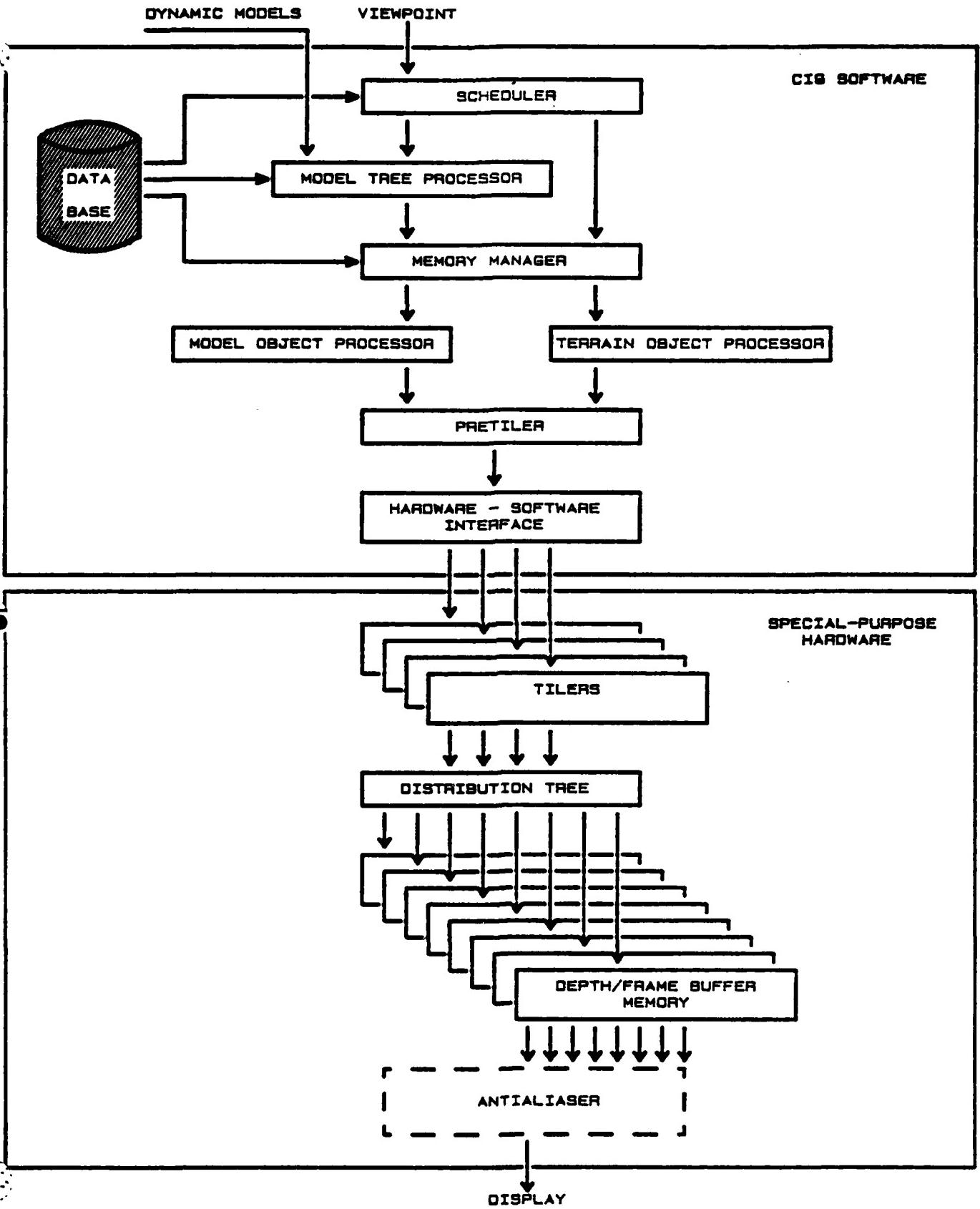
4.4.5 FUTURE DEVELOPMENT

The antialiasing algorithm implemented in this phase was insufficient to prevent aliasing effects in dynamic sequences. Therefore, if this architecture is to be used in high-quality CIG applications, further research on antialiasing techniques is clearly called for (Catm78, Crow81, Feib80, Gupt81). Such research is hampered by the fact that the perception of aliasing effects depends on the update rate, linearity, luminosity, and resolution of the display medium, as well as the lighting conditions under which it is used. Nonetheless, additional Boeing research indicates that a combination of a higher oversampling ratio and larger filters can provide excellent antialiasing. Additional work on the tilers and memory also promise to minimize the cost impact of these more elaborate antialiasing techniques.

Implementation of these new techniques will necessitate a complete redesign of the antialiaser and its interface to the memory. If an interlaced display is being driven, and if the memory is enlarged to accommodate a larger oversampling ratio, the use of delay registers becomes unacceptable. Instead, information from successive rows will have to be read in parallel. In addition, the use of more complicated filters will require more complicated summation circuitry.

5.0 SOFTWARE DESIGN

This section surveys the algorithm development, design, and implementation of the software system of the ARS. This software system was intended to provide image generation capabilities as a front end to the ARS special-purpose hardware. This section begins with a discussion of the visual three-dimensional database that forms the foundation of the software hierarchy. The structure of the database and the methods used to generate it are presented in detail. This section then traces the software hierarchy, starting with the hardware-software interface that connects the software system to the ARS special-purpose hardware. The pretiler, model and terrain object processors, memory manager, and model tree processor are examined next. A discussion of the scheduler, which represents the highest level of the software hierarchy, concludes this section.



5.1 DATABASE CONSTRUCTION

5.1.1 INTRODUCTION

The database construction design takes raw data from many different forms and constructs a hierarchical tree structure that is tailored for maximum image processing efficiency. The hierarchy design permits efficient field-of-view processing along with level-of-detail control. The construction process, using many sources of raw data for terrain and color, enlarges the uses of the gaming area. The modular concept of the node tree allows maximum flexibility in database size, shape, and content. This database design is structured to meet the following criteria and objectives:

- A. To define a simple and logical data structure, allowing easy implementation, maintenance, and growth.
- B. To allow as much flexibility as possible, and to avoid restricting the database on the basis of current processing constraints. This is to permit and encourage experimentation and innovative uses of the DARPA system.
- C. To allow for efficient processing by the DARPA system; specifically, to support efficient field-of-view (FOV) and level-of-detail (LOD) processing on a hierarchically organized database.
- D. To allow the same terrain elevation data (TED) to be paired with a number of color map files; for example, summer and winter scenes of the same location.
- E. To allow very coarse and very detailed TED and color map data to be intermixed in the same database.
- F. To build a self-defining data structure which allows easy upgrades to new versions of the database.
- G. To recognize a number of sources for terrain elevation data and terrain/model coloring.
- H. To allow ability to merge gaming areas for access to extremely large data structures on a modular level.
- I. To provide the ability to quickly generate a viable database using user-friendly menu-driven software.
- J. To provide pinpoint accuracy of terrain location, viewpoint location, and model placement in terms of real world coordinates (geodetic & UTM).

5.1.2 RAW DATA SOURCES

Multiple sources of raw data for terrain modeling and coloring permit maximum flexibility in database construction. Terrain elevation models are created from DMA (geodetic) data, USGS universal transverse mercator (UTM) data, irregular point networks, and manually created elevation models. Coloring information can have multiple sources such as Landsat data, DMA Level V data, USGS land use/land cover data, digitized orthophoto, or synthetic coloring based on terrain features. Other synthetic coloring methods are digitized photographs, shaded relief maps, contour maps, and military inputs such as line-of-sight data and radar.

5.1.3 DATABASE STRUCTURE

5.1.3.1 DATABASE FILE STRUCTURE

The terrain database structure consists of many files, each having a unique purpose and interrelation to the other files (see Figure 41). These files include the gaming area descriptor, consisting of a gaming area header file and a node file. For the terrain elevation and color maps, there exists a matching header file and data file. The models are defined using a model directory file and data files.

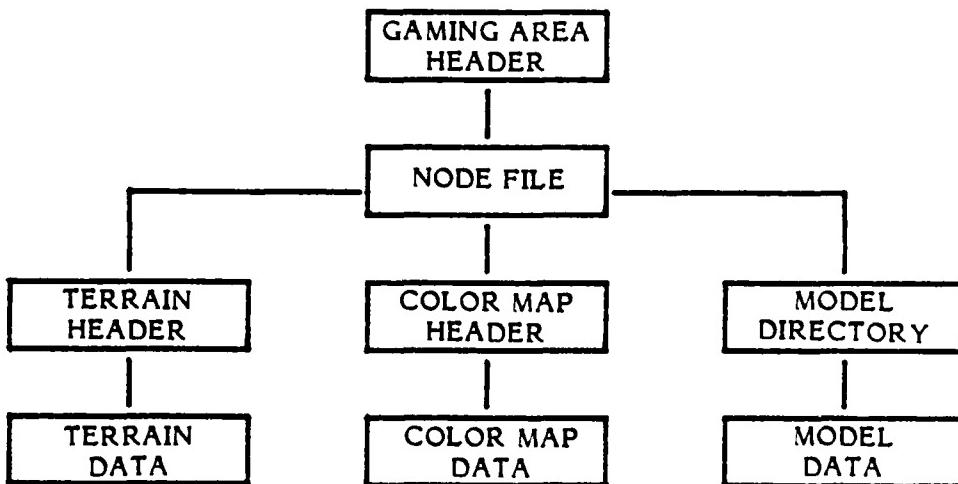


Figure 41. Database File Network

The gaming area header consists of descriptive data for the entire database. Significant parts include: 1) the gaming area centroid in world coordinates, 2) object file names and structure, 3) available color map files and structure. This information is used for initialization of files.

The node file consists of object descriptors in a hierarchical tree structure. These nodes also serve as a link to header files and the model directory. Node descriptors contain object size and location to assist in FOV determination and to facilitate level-of-detail control.

The terrain elevation data files consist of the header file and actual data file. The TED header consists of a pointer to data along with the terrain object size and grid spacing in x and y. The TED data file contains the defining data in a variety of sizes based on the resolution of the node. The data at each vertex point consists of an elevation increment (delta z) from a neighboring vertex and a normalized vertex normal value. The data for each TED node is read using the direct-access method, locating data by the pointer from the TED header.

The color map files contain color map header and data files. The header file contains parameters such as scaling factors and offsets. The color map data is in either a vertex coloring or a mip map format. Vertex color data consists of a one-to-one correspondence of vertices to color array values of (red, green, blue) triplets. The mip map format is structured to use various coloring levels of detail as discussed in Section 6.4.

The model directory file consists of an entry for each static model in the database. This entry specifies the name of the model data file, a model identifier to identify multiple instances of a single model, and an instance identifier to distinguish among multiple instances. The individual model data files contain the tree structure describing the interrelationships of parts in the model, as well as the actual data describing the model parts.

5.1.3.2 TREE STRUCTURE

The design of the terrain database revolves about a hierarchical tree data structure which is determined by the node file. Each part of the terrain database, whether it

be a finite piece of terrain, a model, or a color map, is defined as a node. The significant parts of this node tree are the root, parents, children, siblings and leaves (see Figure 42).

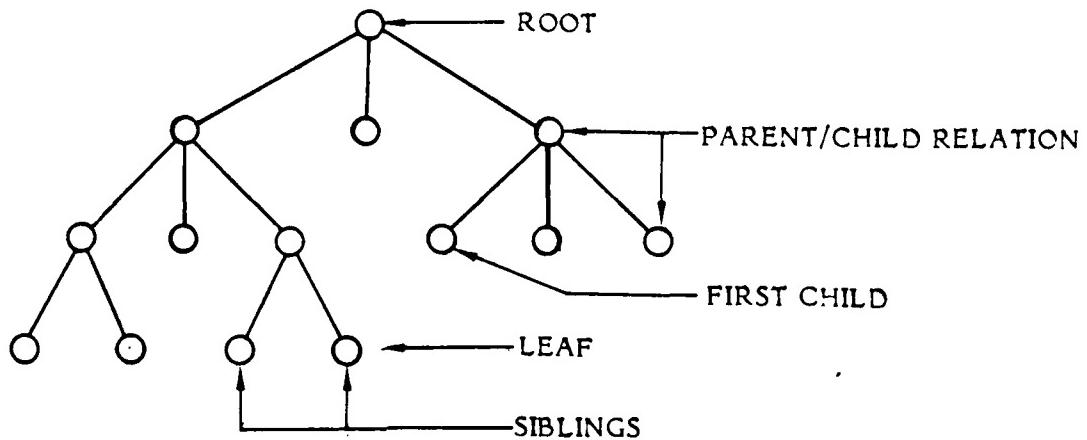


Figure 42. Sample Tree Structure

The root of the tree is the highest level of the hierarchy with its defined centroid location being the centroid of the database. Lower levels of the hierarchy are children with their parents linked directly to them one level above. Siblings are children on the same level with a common parent. A leaf is defined as nodes on the bottom level of the hierarchy thus having no children.

The hierarchical tree node structure is used due to its advantages in image generation. This structure is ideal for conducting the field-of-view (FOV) test and level-of-detail (LOD) control. Both of these processes are directly associated with boosting the speed of image generation.

The FOV test quickly eliminates data outside the field of view. This test requires object location and size information available within the node. The scheduler traverses the node tree from top to bottom, eliminating nodes and all associated children of that node if they are located outside the field of view.

Processing each node in the database at the lowest allowable level-of-detail will speed up image generation. Level-of-detail (LOD) tests based on distance to the node can be accomplished while traversing the node tree. Each node has LOD range values based on the resolution of its data. The scheduler compares distance to the node with these ranges to determine if the node's data should be processed. If the distance and ranges do not match, this indicates that the node's data resolution is insufficient for the level of detail required. The scheduler will then traverse to lower levels (children) with higher data resolution for the terrain node or model.

An asset of the hierarchical tree structure is its inherent flexibility and growth potential. Each node is processed independently with no dependence on children or siblings. This enables objects of various sizes and levels of detail to be nested within a gaming area. A variety of color maps can be associated with a single node for flexibility of color and texture.

The tree structure can grow easily with the addition of models and/or terrain at any level of the tree as a future enhancement for special applications. The entire gaming area tree structure can be thought of as a module. This would enable one to merge adjacent gaming areas for access to large or specially-shaped virtual gaming areas (see Figure 43). This enlarges the scope of gaming area size for continuous overland flight simulation and other applications.

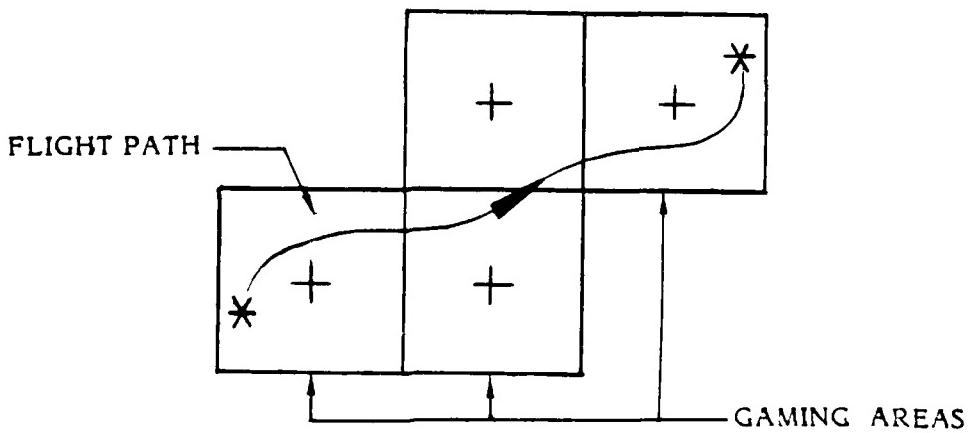


Figure 43. Merging Gaming Areas for Continuous Flight Path

5.1.4 DATABASE CONSTRUCTION PROCESS

The origin of a database is the raw data from which the gaming area is modeled. Certain aspects of the raw data must be considered to determine optimum processing and application of a gaming area database.

5.1.4.1 RAW DATA DESCRIPTION

5.1.4.1.1 RAW TERRAIN DATA

The most comprehensive source of terrain elevation data is the DMA digital elevation models. This data is available for any location worldwide in adjacent data blocks using a regular geodetic grid system. Terrain is also modeled by the USGS using UTM grid system with limited adjacent blocks. Both types are reliable, accessible digital forms and may be used directly to model a gaming area.

An alternative source of elevation data could be from manual digital modeling. This might use sources such as contour maps or photographs as base data. The construction of elevation data using stereo photography results in irregular-point networks requiring resampling before use. Currently, DMA data is the primary source of elevation data due to the contiguous nature of its data blocks and its regular geodetic grid, which requires no resampling.

5.1.4.1.2 RAW COLOR MAP DATA

Color map data can take many forms. The major types are based on:

- A. digitized photographs
- B. hand-modeled colors and patterns
- C. terrain features
- D. Landsat digital data
- E. digital land cover and Level V data

Digitized photographs give the ability to add realistic color and texture to terrain and models. Using corrected photographs such as orthophotos, and precisely

mapping them to terrain, one can show actual features of the terrain such as buildings, roads, and hydrography.

Hand-made color maps are useful for special effects and enhancements to terrain. Different seasonal effect can be shown using snow coloring or fall colors.

Coloring based on actual terrain features can simulate coloring with no base color data. Keying on elevation ranges and slopes of faces, one can simulate natural colorings such as snow, water, and cliffs.

Digital coloring data in the form of Landsat imagery and Landuse polygon fill is used to show actual coloring from vegetation, hydrography or special land use.

5.1.4.1.3 RAW MODEL DATA

Models are defined as 2-D and 3-D solid models. These files are generally created on the Graftek Computer-Aided Design system. A model is defined in its own relative coordinate system. Models are defined as adjoining faces or as a point network to be tiled using Gouraud shading. The raw data at each vertex point includes coordinate values, color intensities in (red, green, blue), and a vertex normal for models to be curve-shaded.

The raw data defining faces will include coordinate points for each face, the color intensity in (red, green, blue) and a face normal.

5.1.4.2 TERRAIN ELEVATION DATA (TED) CONSTRUCTION

5.1.4.2.1 RAW DATA MANAGEMENT

The first step in database construction is to determine the general location and boundaries of the base gaming area in real world coordinates (latitude/longitude). The basic building blocks of a gaming area are 1° latitude x 1° longitude sections. Each gaming area is usually contained within a maximum of four such sections (see Figure 44).

GROUPING
 RAW DMA DATA
 $1^\circ \times 1^\circ$ BLOCKS
 ADJACENT
 (COMMON EDGE)

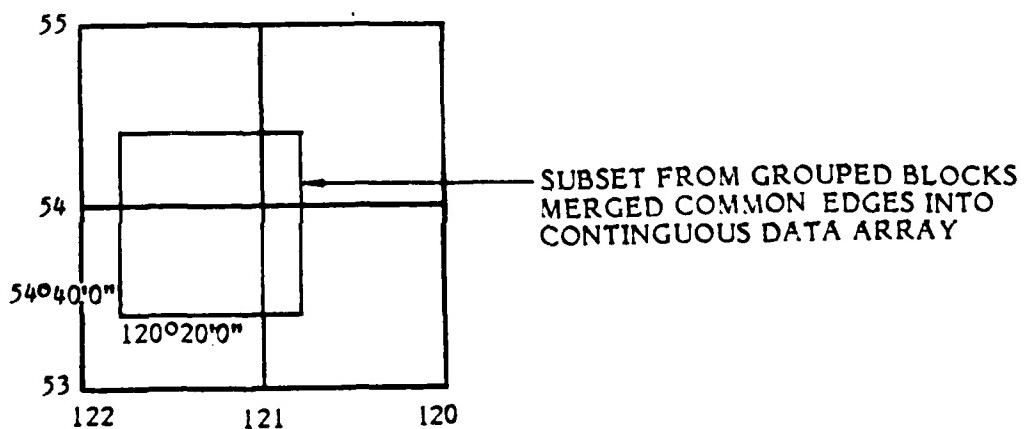
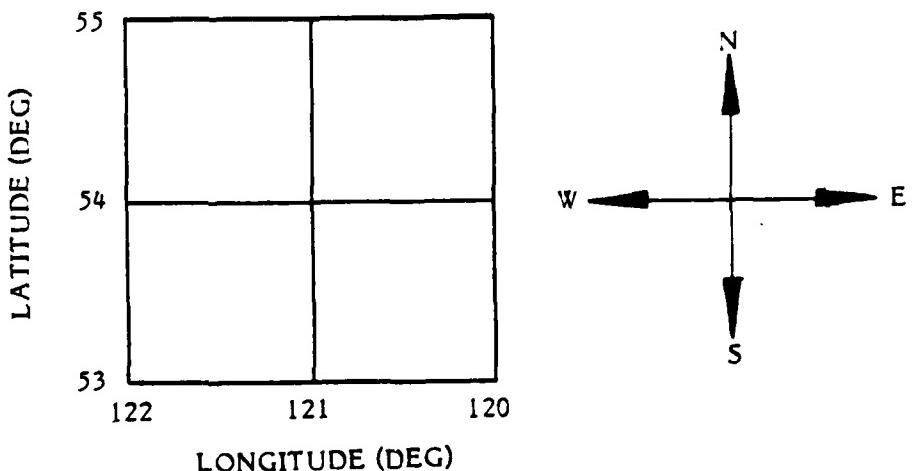


Figure 44. DMA Raw Terrain Data Management

The data blocks encompassing the proposed gaming area are read from tape, carefully tracking precise location of data in latitude/longitude. The data blocks are subdivided and merged as necessary into a single, coherent block of elevation data, and put into a direct-access format (see Figure 44). From this subset of elevation data, a gaming area of any size within the subset's bounds can be made. The next step is the input of the TED leaf node size in number of data points, and the gaming area size in number of leaf nodes in x and y. This defines the gaming area size, and with the centroid selected, the construction of the database files can

begin. The subset block is now subdivided into a raw gaming area array of elevation values required in construction of the data and header files, as shown in Figure 45.

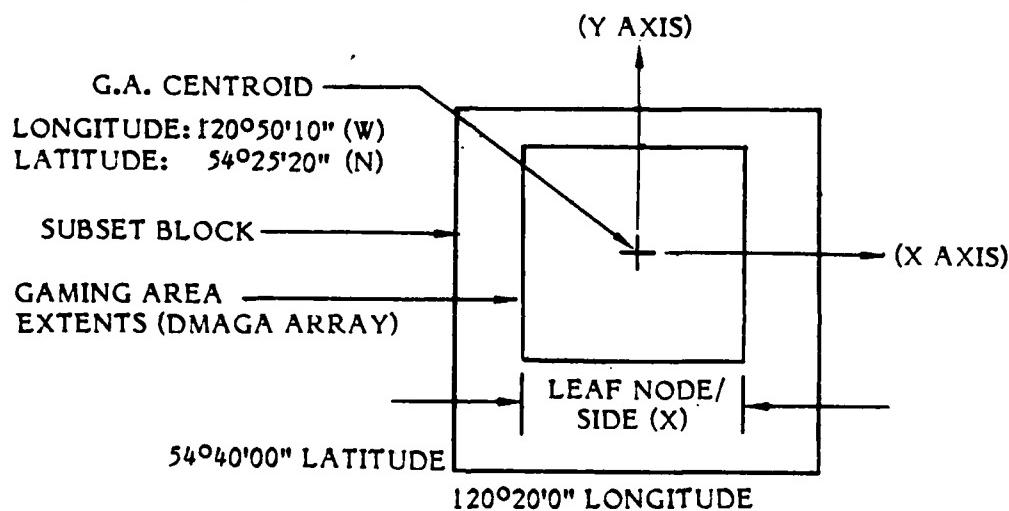


Figure 45. Raw Gaming Area Array (RAWGA)

5.1.4.2.2 TED FILE CONSTRUCTION

The tree structure for the node file is initially created using terrain nodes from the raw gaming area (RAWGA) array along with the leaf node information. The TED node tree is tailored into a quadtree format. This eases construction and speeds field-of-view processing. This structure breaks each parent node into four quadrants (children). The nodes at higher levels of the tree represent clusters of leaf nodes (see Figure 46).

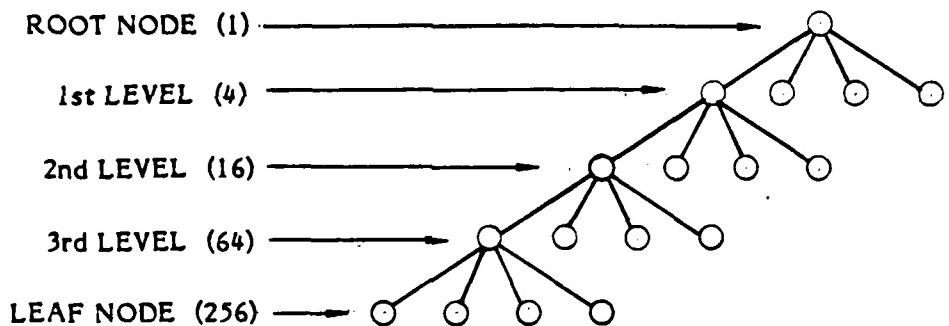
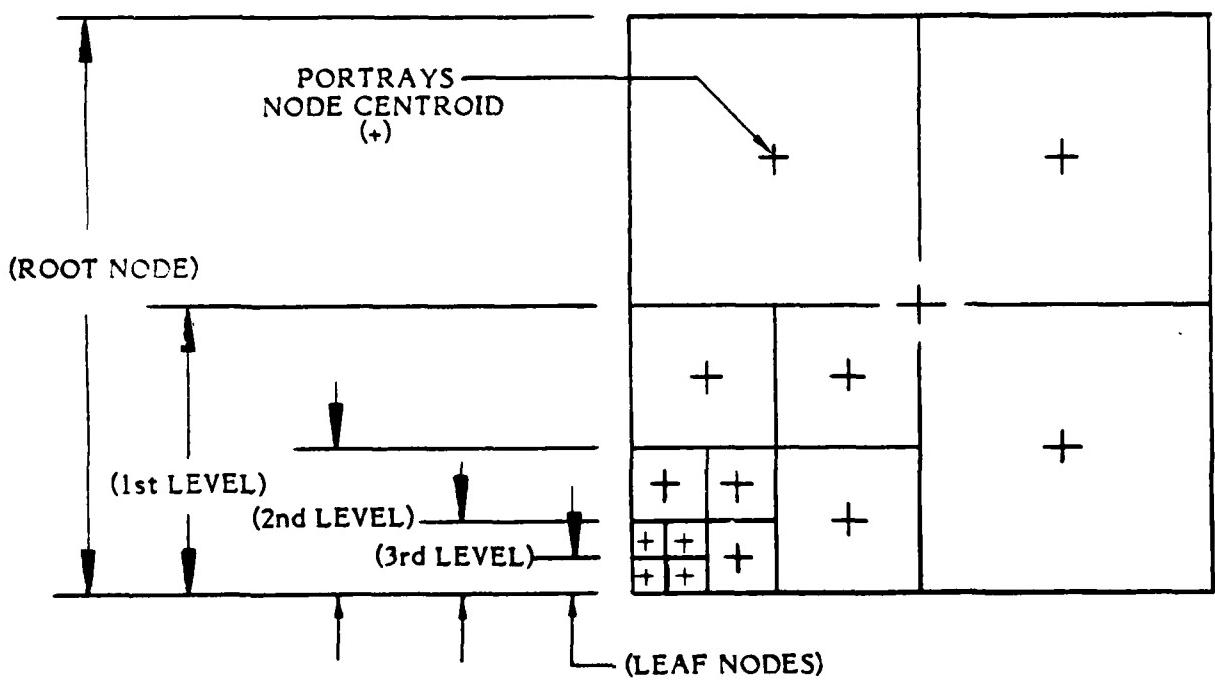


Figure 46. Sample Gaming Area and TED Quadtree

Significant parameters calculated in node construction are the following: 1) data type, 2) pointers to data header, first-child, and sibling, 3) child number, 4) centroid location as an offset from the parent, 5) radius, 6) average surface normals, and 7) level-of-detail ranges.

The data type identifies a node to be a terrain, color map, model, or cluster node. The pointer to the data header is the record number associated with that node in the data header file. This pointer is zero for cluster nodes. The sibling pointer is a record number of a child's sibling in the node file, which is zero for the last child of

a parent. The pointer to the first child of a parent is the record number of the first child's node in the node file.

The node centroid location is defined as the offset of the node's centroid with respect to its parent's centroid. The radius is the minimum outside radius encompassing the node (see Figure 47). The absolute offset in x and y for all nodes on the same level will be equal in a quadtree structure. The sign of the locating offset is set according to the node quadrant based on child number (1 - 4) as shown in Figure 47.

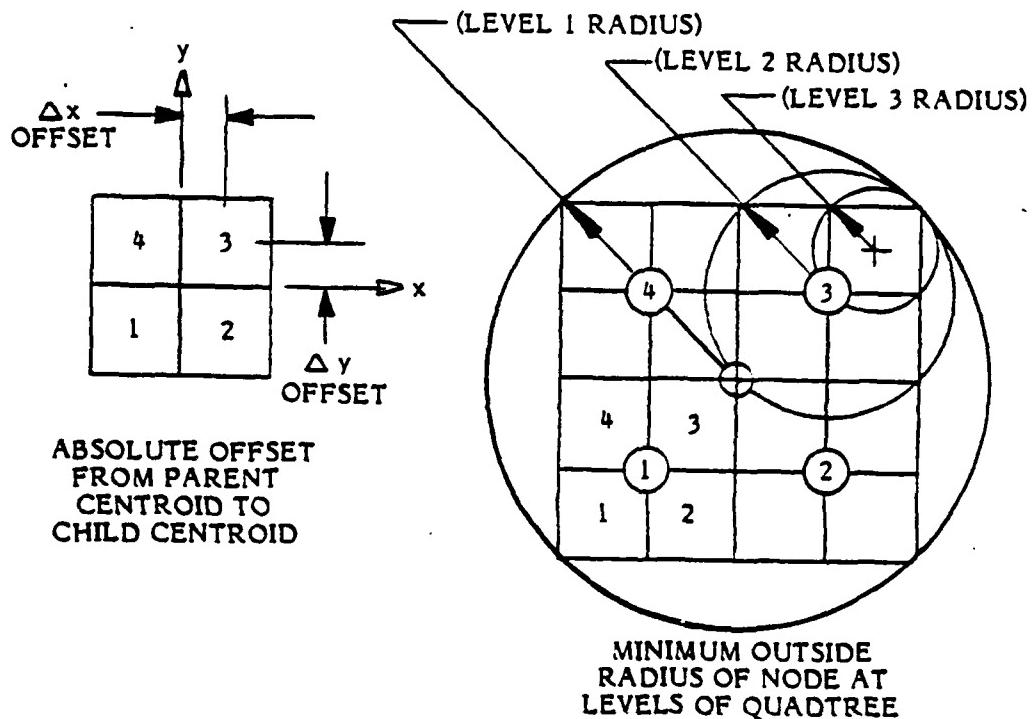
The level-of-detail ranges and average surface normals are used to determine whether this node is at an appropriate level of detail to be processed. They allow level-of-detail transitions based on position as well as orientation relative to the viewer.

The TED header file contains the size of the node in number of vertices, and spacing of the vertices in x and y. The spacing of vertices is in arcseconds when using DMA data. The last main item in the header is the data block pointer. This pointer is to the appropriate data sector in the TED data file.

The size of the TED data node is based on the leaf node size as input by the user and the level of the node on the tree. The defining data for a TED node includes two arrays. They are the incremental elevation (delta z) array, and the vertex normal (VNORM) array.

5.1.4.2.3 DELTA-Z ARRAY

The delta-z array is a way to represent actual elevation values (z) as elevation increments from one vertex to another within a node. This structure of elevation storage is used to reduce computation for the back-face elimination test and coordinate transfer processes.



	CHILD #			
	1	2	3	4
x	-	+	+	-
y	-	-	+	+

OFFSET SIGN
BASED ON CHILD #

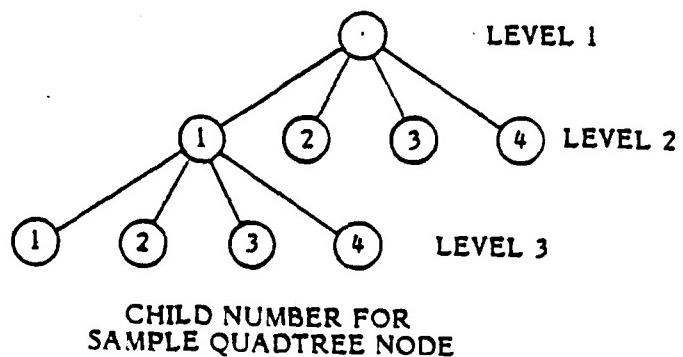


Figure 47. Node Variable Development

The (0,0) element of the array is the only vertex in the delta-z array with a true z value (see Figure 48). The elements of the delta-z array in the left-most column contain the change in altitude (delta z) relative to the vertices to their south.

$$\text{delta } z(0,j) = z(0,j) - z(0,j-1) \quad \text{for } j = (1 \rightarrow n)$$

The elements in all other columns contain the change in altitude relative to the vertices to their west.

$$\text{delta } z(i,j) = z(i,j) - z(i-1,j) \quad \text{for } i = (1 \rightarrow n) \\ \text{and } j = (1 \rightarrow n)$$

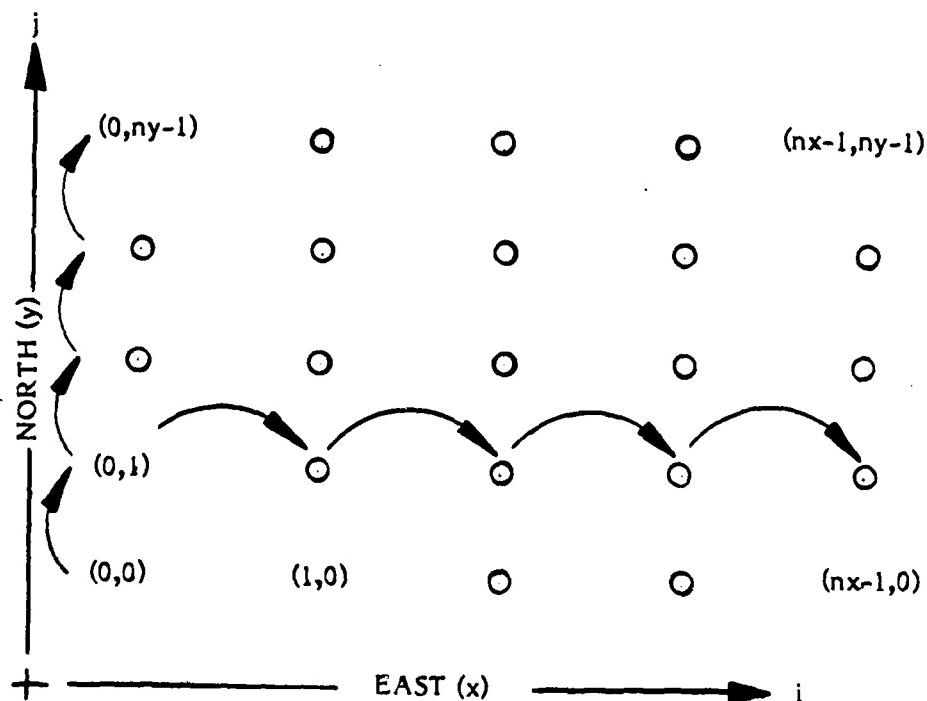


Figure 48. Delta-Z Array

5.1.4.2.4 VERTEX NORMAL ARRAY

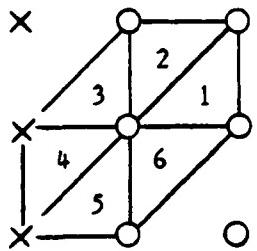
The vertex normal array assigns a normalized vertex normal value to each vertex. The array size will equal the number of vertices of the node in x and y. To calculate the vertex normal, data is required to define all triangles surrounding the vertex.

Therefore, the raw data required for a $(n \times m)$ vertex normal array is a $(n+2) \times (m+2)$ array of raw elevation data as shown in Figure 49.

The vertex normal is calculated as a weighted sum of the face normals of the six surrounding triangle faces. The derivation of this process is shown in Table 11.

Table 11. Vertex Normal Derivation
(Continued on following page)

6 surrounding triangle faces as tiled around a vertex

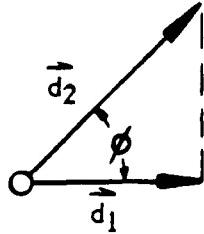


1 \vec{d}_1, \vec{d}_2 : vectors defining surface

2 \hat{n} : normalized face normal

$$\hat{n} = \frac{\vec{d}_1 \times \vec{d}_2}{|\vec{d}_1 \times \vec{d}_2|}$$

$$\hat{n} = \frac{\vec{d}_1 \times \vec{d}_2}{|\vec{d}_1 \times \vec{d}_2|}$$



3 A_i : Area of triangle surface

$$A_i = 1/2 |\vec{d}_1 \times \vec{d}_2|$$

4 D_i : Distance to vertex from center of triangle (surface).

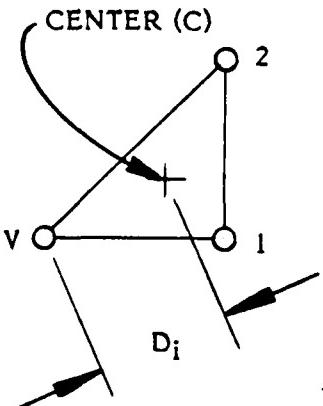
$$C_x = (x_1 + x_2 + x_v) / 3$$

$$C_y = (y_1 + y_2 + y_v) / 3$$

$$C_z = (z_1 + z_2 + z_v) / 3$$

$$D_i = (x_v - x_C)^2 + (y_v - y_C)^2 + (z_v - z_C)^2$$

Table 11. Vertex Normal Derivation
(Concluded)



5 VN: weighted vertex normal

$$VN = 1/6 \sum_{i=1}^6 \left(\frac{A_i}{D_i^2} \right) \hat{n}_i$$

$$\vec{VN} = 1/6 \sum_{i=1}^6 \left(\frac{1/2 |\vec{d}_1 \times \vec{d}_2|}{D_i^2} \right) \frac{\hat{n}_i}{|\vec{d}_1 \times \vec{d}_2|}$$

$$\vec{VN} = 1/12 \sum_{i=1}^6 \frac{\hat{n}_i}{D_i^2}$$

6 \hat{VN} : normalized vertex normal

$$\hat{VN} = \frac{\vec{VN}}{|VN|}$$

where $|VN| = VN_x^2 + VN_y^2 + VN_z^2$

$$VN_x = \sum_{i=1}^6 \frac{N_{ix}}{D_i^2}$$

$$VN_y = \sum_{i=1}^6 \frac{N_{iy}}{D_i^2}$$

$$VN_z = \sum_{i=1}^6 \frac{N_{iz}}{D_i^2}$$

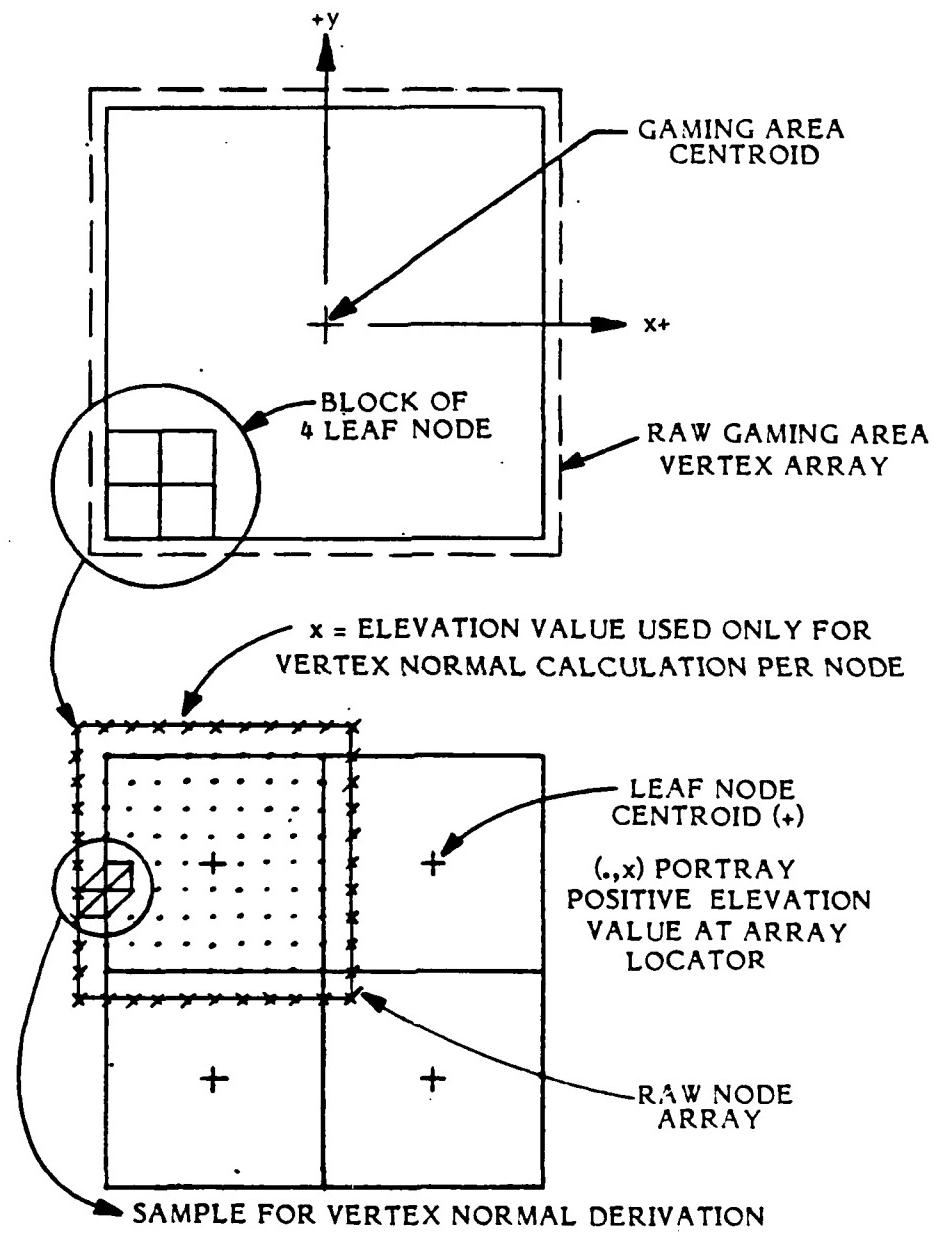


Figure 49. Required Raw Data for Vertex Normals

5.1.4.3 COLOR MAP FILE CONSTRUCTION

The color map files include a header and a data file. The header parameters include the offsets, size and scaling factors of the color map in the x and y directions, and the data block pointers. The offset parameter is used to locate the map relative to the terrain and is calculated as the terrain object data is constructed. The size and scaling factors are set by the user when assigning color maps to terrain objects. The data block pointers point to the correct data sector in the color map data file.

The color map data file holds blocks of color data; it is associated with each terrain node by a sibling relationship with a TED node. This data could be a vertex color array of (red, green, blue) triplets, or a mip map array of color values.

A vertex color array associates a color value directly with each vertex in the terrain data model. The construction process could, for example, assign a selected color to a vertex based on altitude or the vertex normal value to simulate natural coloring of water, cliffs, snow, and other seasonal effects based only on terrain elevation data.

The color data for a mip map contains color triplets defining the map for many levels of detail. The database construction process is independent of this design process. The mip map can have inputs from digitized photos, digital color data, or selected colors and patterns.

The model files consist of a directory file and a model data file for each model in the gaming area. The detailed entry for each model in the directory includes filename of the model, the model identifier, and the instance of the model. The model data files are created in a separate process. The defining data is usually a group of polygons, defined by vertices relative to the origin of the model axis system.

5.1.5 FUTURE DEVELOPMENTS AND ENHANCEMENTS

The major improvement to the database construction process would be the addition of special-purpose software. This software would expand the raw data inputs and

enhance the flexibility of the shapes and structure. An overview of these objectives for the software routines is discussed below:

A. Interactive database modification—

The objectives of this routine would be:

1. Addition or substitution of color maps.
2. Model placement directly on terrain, based on x,y location.
3. Addition or substitution of terrain objects of similar or different data resolution.

B. Increase flexibility for gaming area size and shape—

The size and shape of the nodes is controlled in part by the level-of-detail control process. With this process revised, restrictive code could be removed to increase flexibility for gaming area size and shapes.

C. Read and construct database from USGS terrain elevation models—

This raw data management routine would read terrain elevation data provided by the USGS in UTM coordinates from standard tape formats.

The data would be grouped into a raw gaming area array format suitable for the database construction process.

D. Generation of flightpath datafile of viewpoints—

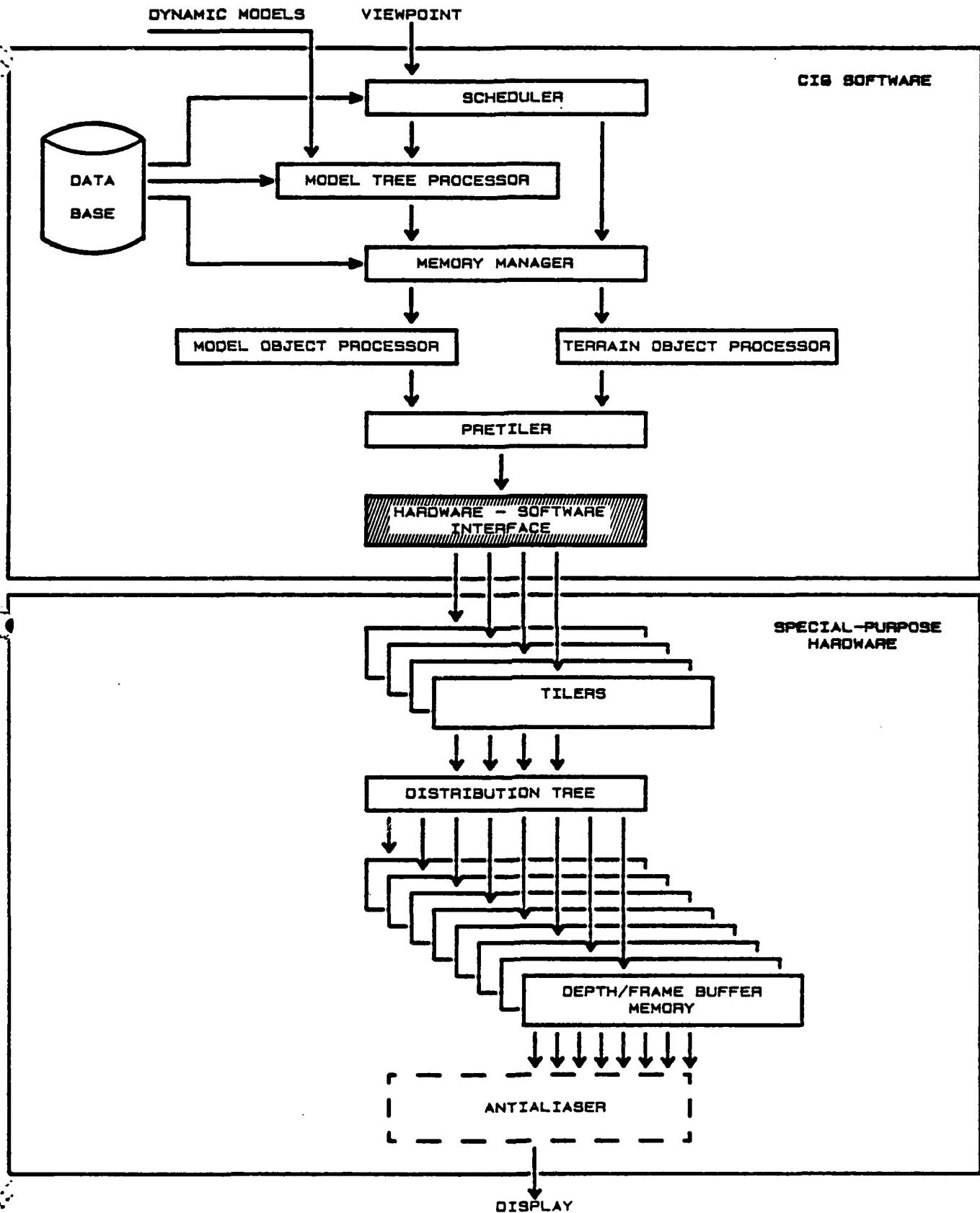
This software would generate a series of viewpoints along a generated flightpath for a gaming area. The inputs would include initial and periodic flight locations, flight speeds, and initial speed magnitude.

E. Model construction of terrain—

This tool would take irregular point networks defining terrain and construct mathematical surface-defined models. This could use a Bezier curve representation as discussed in Section 6.2.

F. Texture mapping of orthophotos—

The objective of this routine is to map photographs, corrected for the curvature of the earth, to the terrain. This involves accurate location of the digitized photos onto terrain objects. The result would match terrain photographs with terrain contours to simulate actual features and land uses.



HARDWARE-SOFTWARE INTERFACE

5.2 HARDWARE-SOFTWARE INTERFACE

The hardware-software interface (HSI or interface) provides communication between the hardware architecture research system and the image generation software. The interface receives triangle data from the pretiler software, formats the data so that the ARS will recognize it, and transfers the data from storage when appropriate. The interface also handles all control signals required by the ARS.

Communication with the ARS occurs through two different communication paths. (Refer to Figure 50.) The first path is the hardware control/status board (CSB), which is the sequence controller for the hardware. The second path is four I/O channels which communicate triangle data to the four hardware tilers.

The ARS requires a strict chain of communication events to occur with the software to enable it to function properly.

- A. System reset - If this is the first frame, the CSB is signaled by the software to reset all hardware.
- B. Clear interrupt - The software sends a clear interrupt signal to the CSB resetting an interrupt flag from the previous image.
- C. Triangle transfer - The software sends triangle data to the appropriate hardware tilers.
- D. I/O complete - Once the software has completed transferring its triangle data, it notifies the CSB that the frame is finished and processing of the next frame may begin.
- E. Genisco interrupt/transfer - When the CSB determines that the hardware has finished processing, it interrupts the Genisco Graphics Processor and transfers the image to the Genisco frame buffer for display.

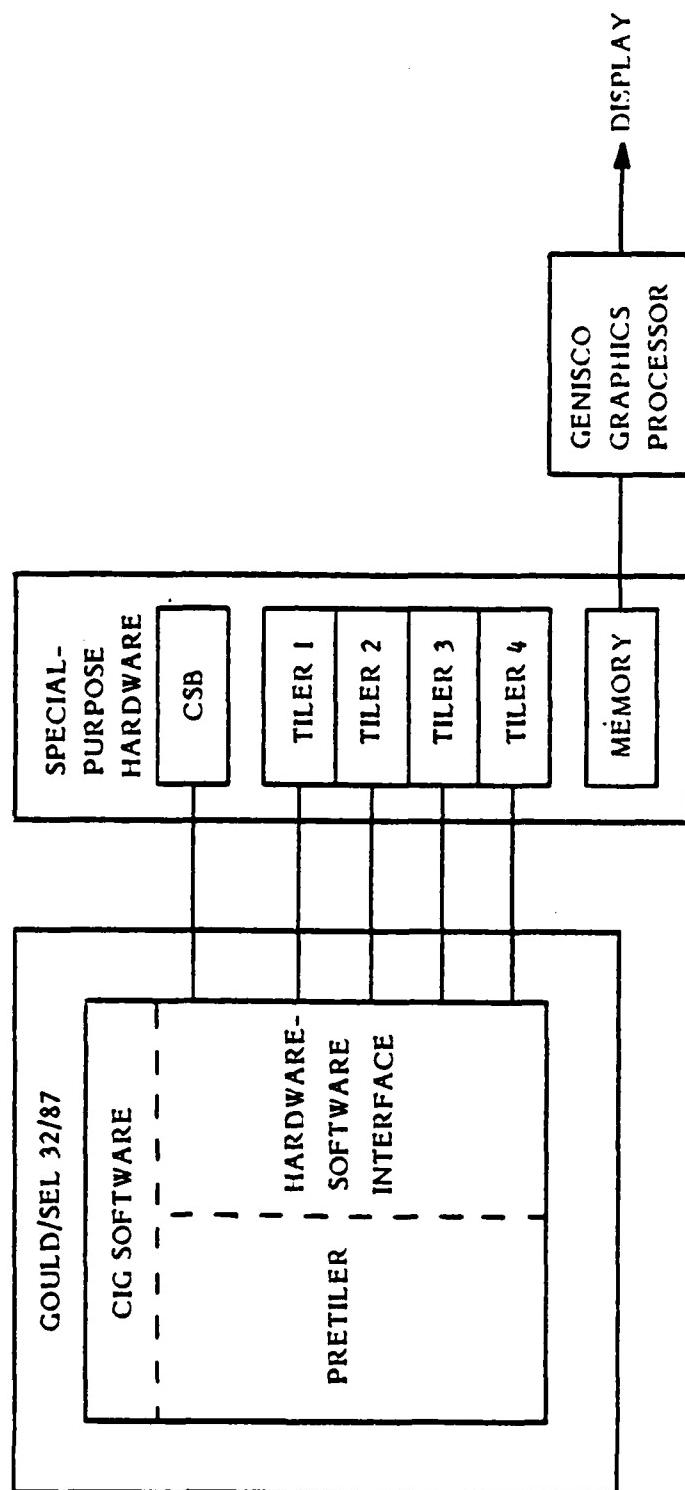


Figure 50. Hardware-Software Interface Communication Channels

F. SEL interrupt - After the hardware memory finishes the transfer, the CSB interrupts the SEL computer, setting a flag which indicates to software that the memory transfer is complete. It may then resume the sequence above, if desired.

The triangle data required by the hardware tilers is provided by the pretiler. It must be organized into the proper sequence and converted into formats the hardware tilers recognize. The conversion is necessary due to differences in number formats between the Gould/SEL 32/87 and the ARS. The ARS recognizes three formats: 16-bit fixed-point, 16-bit floating-point, and 32-bit floating-point. However, the floating-point formats of the SEL and ARS differ. Below is a description of the data required by the ARS tilers for one triangle. Refer to Figure 51 for triangle word formats.

The first three words of data for a triangle contain the screen i,j vertex information for the triangle. The vertex information is formatted as 16-bit fixed-point representations for i and j such that i is in the most significant two bytes and j is in the least significant bytes. The three words are arranged so that the minimum i is in word one and the maximum i is in word three.

As previously described in Section 4.2.2.6, any triangle enclosed in a circumscribing rectangle will have at least one vertex in a corner of the rectangle. This vertex will be referred to as the corner vertex. Word four contains the 16-bit floating-point color value for the corner vertex in the least significant bytes. Bit 16 of this word contains the background flag, which the memory uses to initiate automatic-write cycles. Bit 17 is the under/over sample flag which sets a sampling tolerance level in the tiler. Bits 18-19 specify in which corner of the circumscribing i-j min-max rectangle the corner vertex lies (see Figure 52). The corners are numbered as follows: 0 - upper left, 1 - upper right, 2 - lower left, 3 - lower right. Note that it is possible to have multiple corner points whenever a triangle edge is horizontal or vertical, either of which is acceptable.

Word five contains 16-bit floating-point delta-color information for i and j. Delta-color is the amount of color change from one pixel to the next in the i or j direction. Delta-color in the j direction is in the most significant bytes, while delta-color in the i direction is in the least significant bytes.

WORD		16	16			
1	I (1)		J (1)			I, J's ORDERED SUCH THAT I (1) < I (2) < I (3) FIX PT. 16 BIT NUMBERS
2	I (2)		J (2)			
3	I (3)		J (3)			
4	12	2	1	1	16	FLOATING PT. 16 BIT NUMBERS
4	UNUSED	CORNER	U/O	BG	CO	
5	16				16	
5	DCDJ				DCDI	
6					32	
6					DO	
7					32	
7					DDDI	FLOATING PT. 32 BIT NUMBERS
8					32	
8					DDDJ	

Figure 51. Data Format for One Triangle

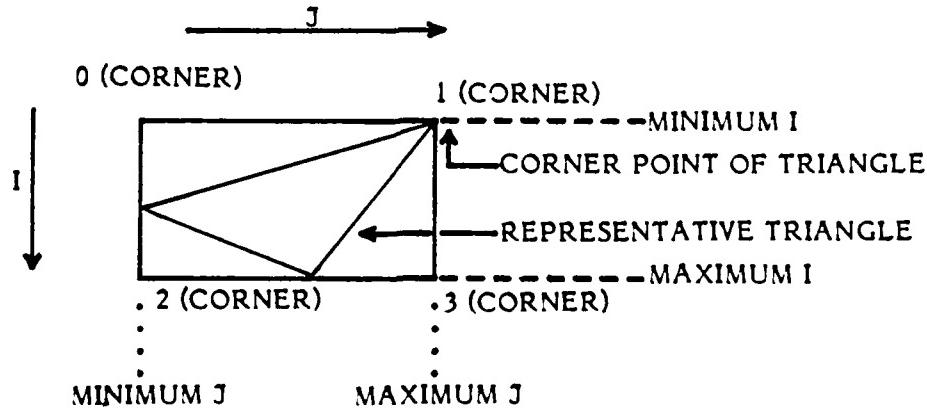


Figure 52. Corner Definition

Words six through eight are ARS 32-bit floating-point numbers containing depth information. Word six is the depth of the corner vertex. Words seven and eight are the delta-depth in the i and j directions. Like delta-color, delta-depth is the amount of depth change from one pixel to the next in the i or j direction.

The hardware-software interface was designed to keep all four tilers active simultaneously. To accomplish parallel processing by four tilers, no-wait I/O, semaphores and N-buffering were utilized.

The idea of no-wait I/O is very simple. During normal I/O, processing ceases until the I/O process has completed because the task presumably needs the data (input) or the memory (output) for further computation. No-wait I/O allows the task to continue computing while the I/O transfer occurs so that the CPU does not remain idle. The task is responsible for not using or modifying anything affected by the I/O transfer until the transfer is complete. The HSI uses this feature to initiate triangle transfer to one hardware tiler while it computes and initiates another transfer to the next hardware tiler. The intended result is to maintain parallel processing in the ARS tilers.

To prevent the task from utilizing a tiler that is still busy with an I/O transfer, a semaphore is set for each tiler. A semaphore is a signal or flag indicating the state of whatever condition the semaphore controls. The three possible states indicated by this flag are busy, available, and unavailable. When an I/O transfer is initiated, the corresponding tiler semaphore is set to the busy state. Upon return from an I/O transfer, the tiler semaphore is reset to available. A tiler may only initiate transfer when in the available state. When the tiler is unavailable, it indicates the tiler may not be used for a variety of reasons; for example, it may be out for repair or a cable may be malfunctioning. The task then knows not to use this tiler.

The concept of N-buffering is an expanded case of double buffering. Triangle data will be in transit to four tilers at once, and the buffers being sent cannot be modified until the I/O completes. Additional buffers are required to store triangles, thus allowing the task to continue processing. The number of required buffers, N, is dependent on the ratio of the software processing rate to the hardware tiler processing rate. For images with any degree of complexity, this ratio is very low, since the hardware is much faster than the imagery software. A

value of N=5 is sufficient to allow one buffer to be transferring data to each of the four tilers with one buffer for storing newly-processed triangles. However, to measure tiler statistics, a much larger N was chosen to take up all available memory. Then, a frame of triangles could be precomputed, stored in N-buffers, and transferred to the tilers as fast as possible. The buffer size was made variable to enable "tuning" the software to the most suitable size to keep the four hardware tilers busy.

Each of the buffers requires a semaphore similar to the tiler semaphores. The potential states are empty, filling, full, and transferring. Normally, a buffer will start as empty and change to filling as the triangles are loaded. Once full, it will be passed to a tiler and the status is changed to transferring. Upon return from I/O, the buffer is flagged as empty (see Figure 53). The semaphores also aid in error detection. If a buffer or tiler is found in the wrong state for a particular process, diagnostic messages aid in locating the problem.

Figure 53 shows the interaction of buffers and tilers. In the event a buffer becomes full and no tiler is available, the buffer waits for a tiler to complete. When a tiler completes, it releases its buffer and takes the next full one. Otherwise, it waits for a buffer to become full. The tilers and buffers are not necessarily FIFO queues since not all buffers require the same amount of tiler processing time.

When the software completes processing, the HSI waits for the four tilers to return from no-wait I/O before signaling the CSB that the SEL is finished. Upon hardware completion, the ARS interrupts the Genisco Graphics Processor and dumps the frame to the frame buffer. An interrupt from the hardware indicating that it has finished transferring the frame, and an acknowledgment by the HSI via a clear interrupt signal to the CSB complete the chain of events necessary to produce a computer-generated image.

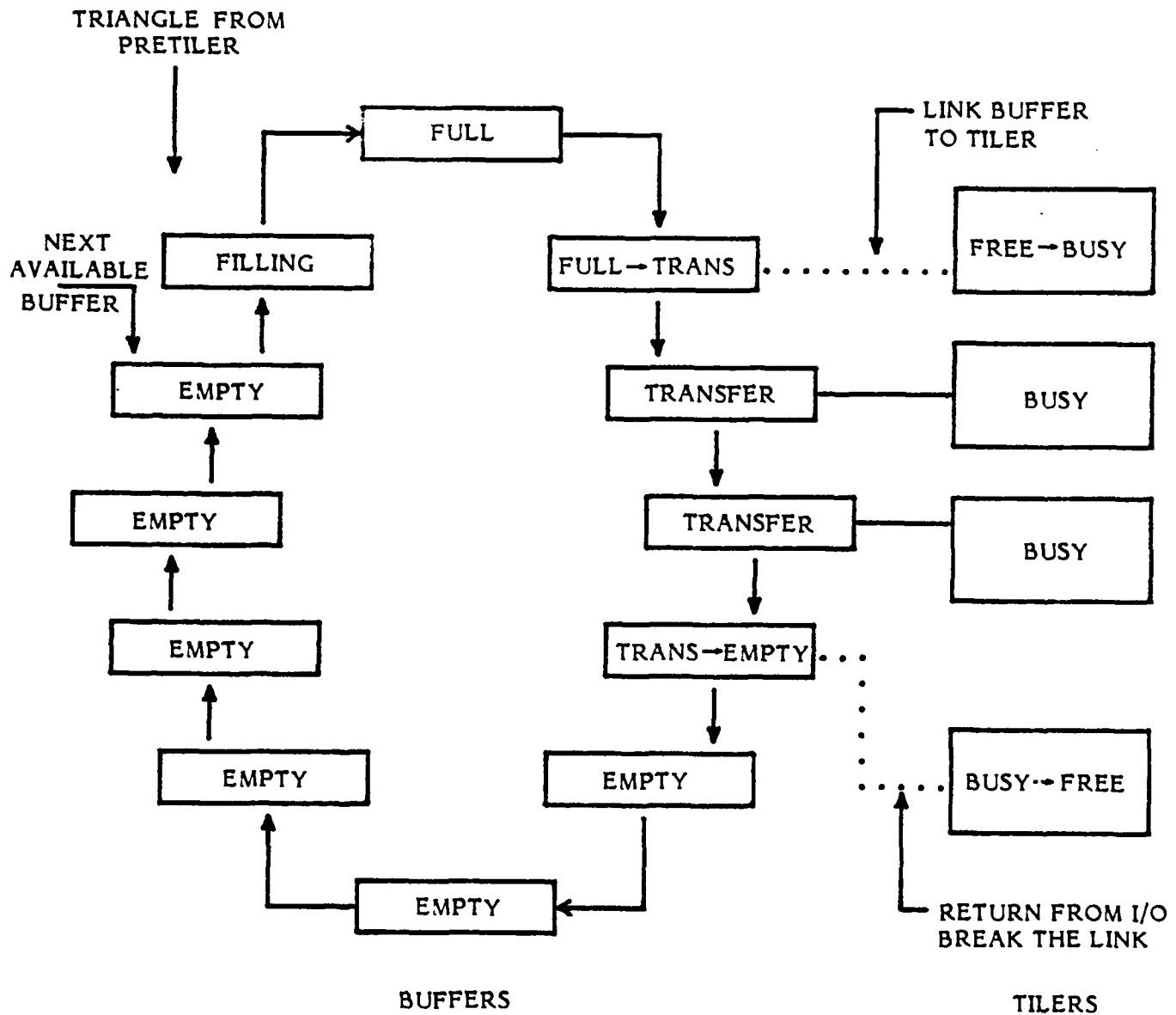
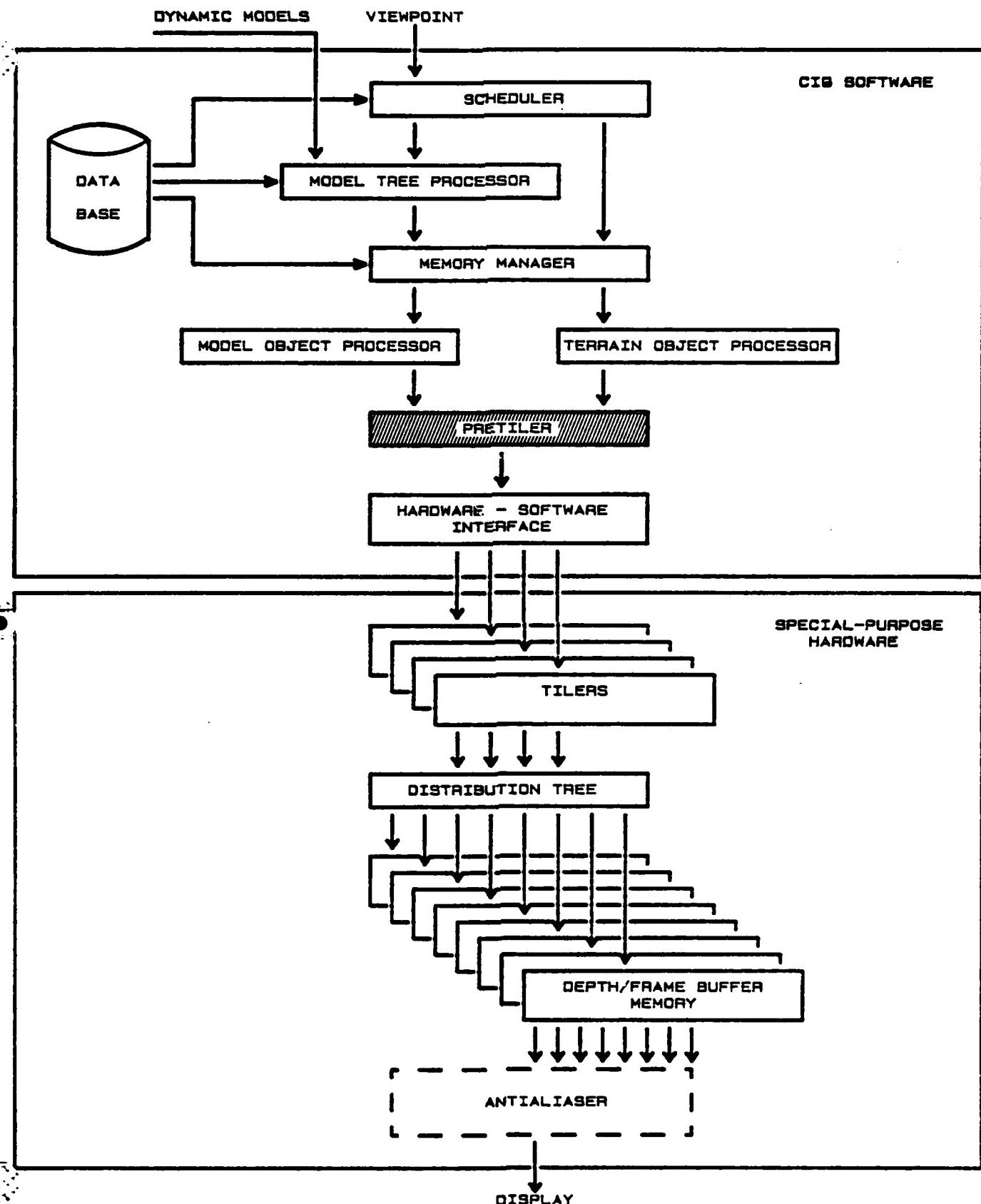
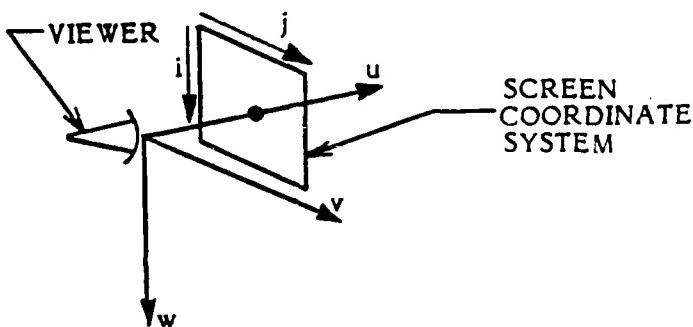


Figure 53. Buffer/Tiler Interaction



5.3 PRETILER

The pretiler receives three-dimensional triangle data in viewpoint space from the terrain and model object processors, and computes the screen-space information needed by the tilers (see Figure 54). Input to the pretiler comes in the form of viewpoint coordinate vertices, a color, and a unique number for each vertex of a triangle. The pretiler clips a triangle in the U direction, projects it to screen coordinates, and clips it to the screen to form an n-sided polygon. This n-sided polygon is subdivided into $(n-2)$ triangles. Coloring information and other variables required by the tiler are then calculated.



Note that $u=1$, $v=0$, $w=0$ is the center of screen space and that the i,j plane is parallel to the v,w plane.

Figure 54. Spatial Orientation of Viewpoint and Screen Space Coordinates

A great deal has been written on the subject of clipping (Newm79, Spro68, Cyru78). The clipping algorithm used by the pretiler most closely resembles Sutherland and Hodgman's algorithm (Suth74a). It clips the entire polygon to each boundary in sequence, rather than each separate edge against all boundaries simultaneously. The Sutherland/Hodgman algorithm clips in 3-space, with clipping performed on each boundary plane in succession. In contrast, the pretiler performs two stages of clipping: the hither and yon planes in 3-space, and the screen boundary in 2-space.

In the first stage, the pretiler clips to the yon plane (the plane past which the viewer cannot see) by discarding any triangles that are completely beyond it and keeping those that are fully or partially before it (see Figure 55).

The pretiler then clips to the hither plane (the plane before which the viewer cannot see). The hither clip is necessary to remove points behind the observer that would project incorrectly onto the screen in the perspective computations. Triangles that cross the hither clipping plane are clipped at that point, and the visible sections are calculated and processed further.

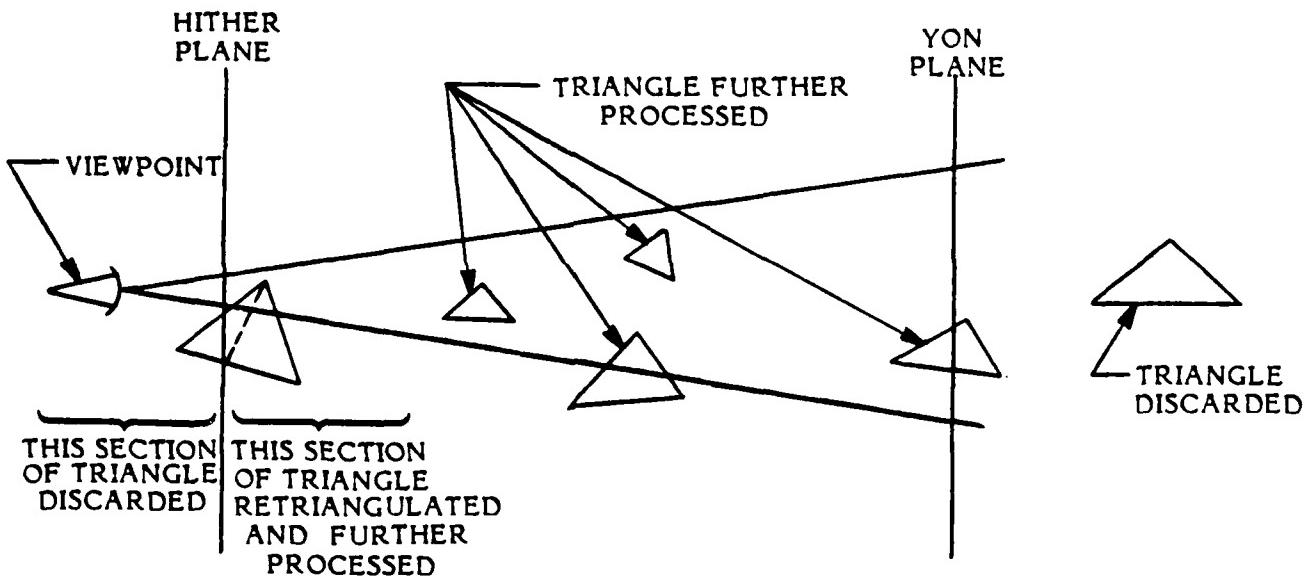


Figure 55. Hither and Yon Clipping

Clipping a triangle to a plane is a simple process. Refer to Figure 56 for the following discussion. Each side of the triangle is tested to see if it crosses the bounding plane. If it does, the point of intersection of the edge and bounding plane is calculated and a new vertex is created. The depth and color for this new vertex is calculated by linear interpolation between the depth and color values of the adjoining original vertices. As a result of clipping, the triangle becomes either a smaller triangle or 4-sided polygon which is then triangulated and passed on for further processing.

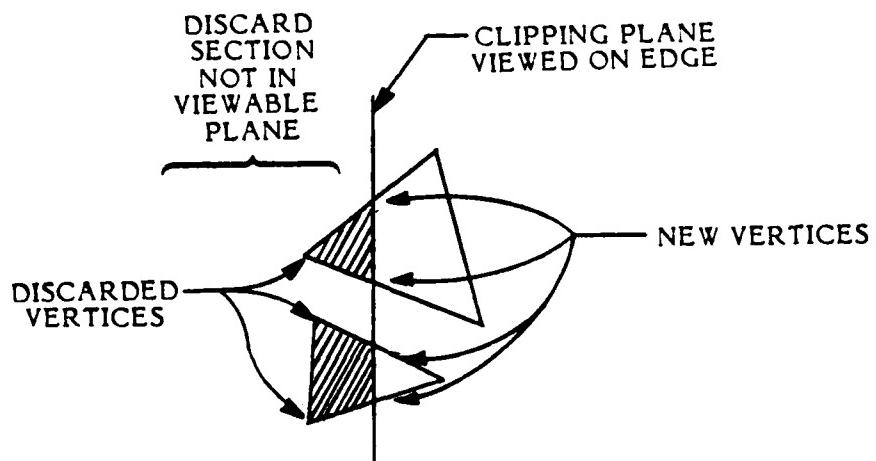


Figure 56. Hither Clipping

At this point, the pretiler projects the triangles from viewpoint space to screen space. For each vertex of a triangle, there is a corresponding vertex number. When a vertex is projected, its projection and vertex number are stored. If another projection of the same vertex is required, the stored projection is used rather than recalculated. On the average, the stored vertex projection value is used two or three times, saving 70% of all projection calculations.

For each projected triangle that lies totally or partially within the screen boundaries, a vertex is chosen as an origin, and its depth and color values are established as base values. Using these base values in conjunction with the color and depth values at the remaining two vertices, color and depth gradients across the triangle are calculated. Because true depth and color are not linear in 2-space, these calculations express depth in a linear form and allow color to depend on the variation in depth (see Section 4.2). Projected triangles are then clipped to screen boundaries using the depth and color gradients to compute the depth and color of new vertices (see Figure 57).

The Sutherland/Hodgman algorithm differs from the pretiler algorithm in that it performs all clipping in 3-space and then projects to 2-space. A triangle clipped in 3-space becomes an n -sided polygon which is triangulated into $(n-2)$ triangles.

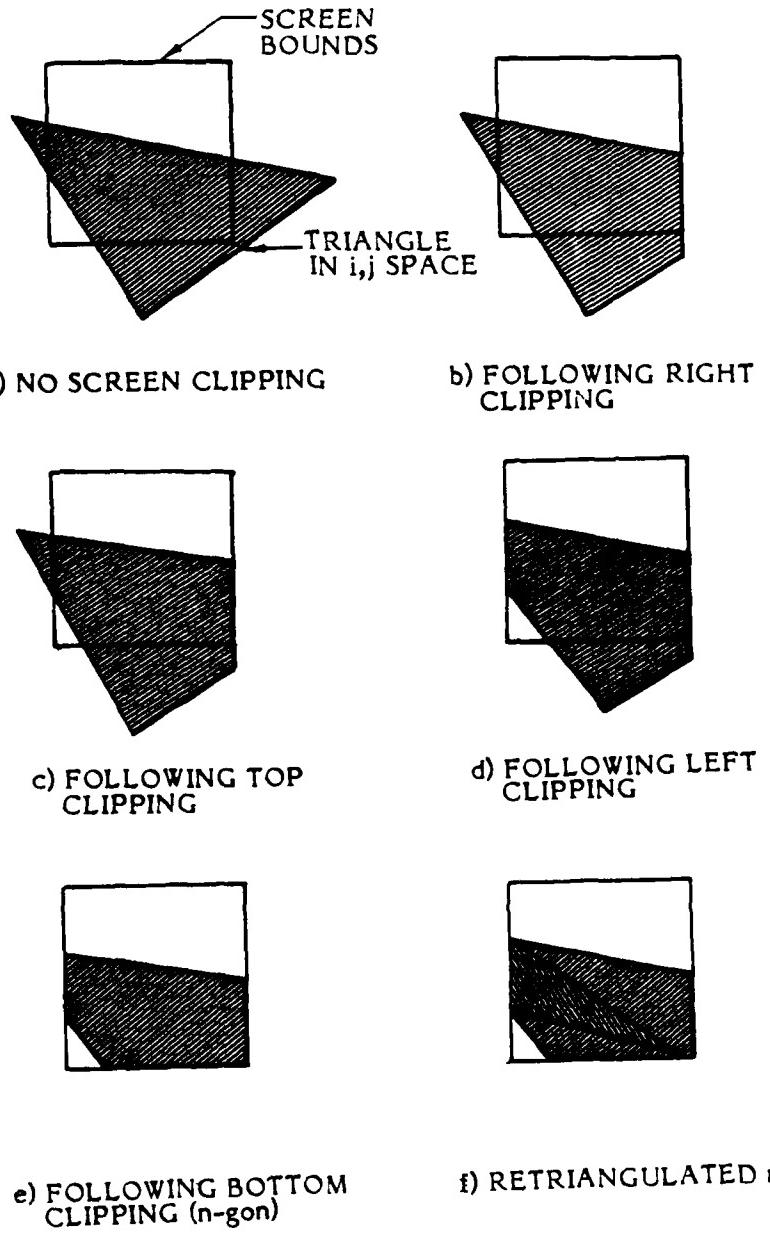


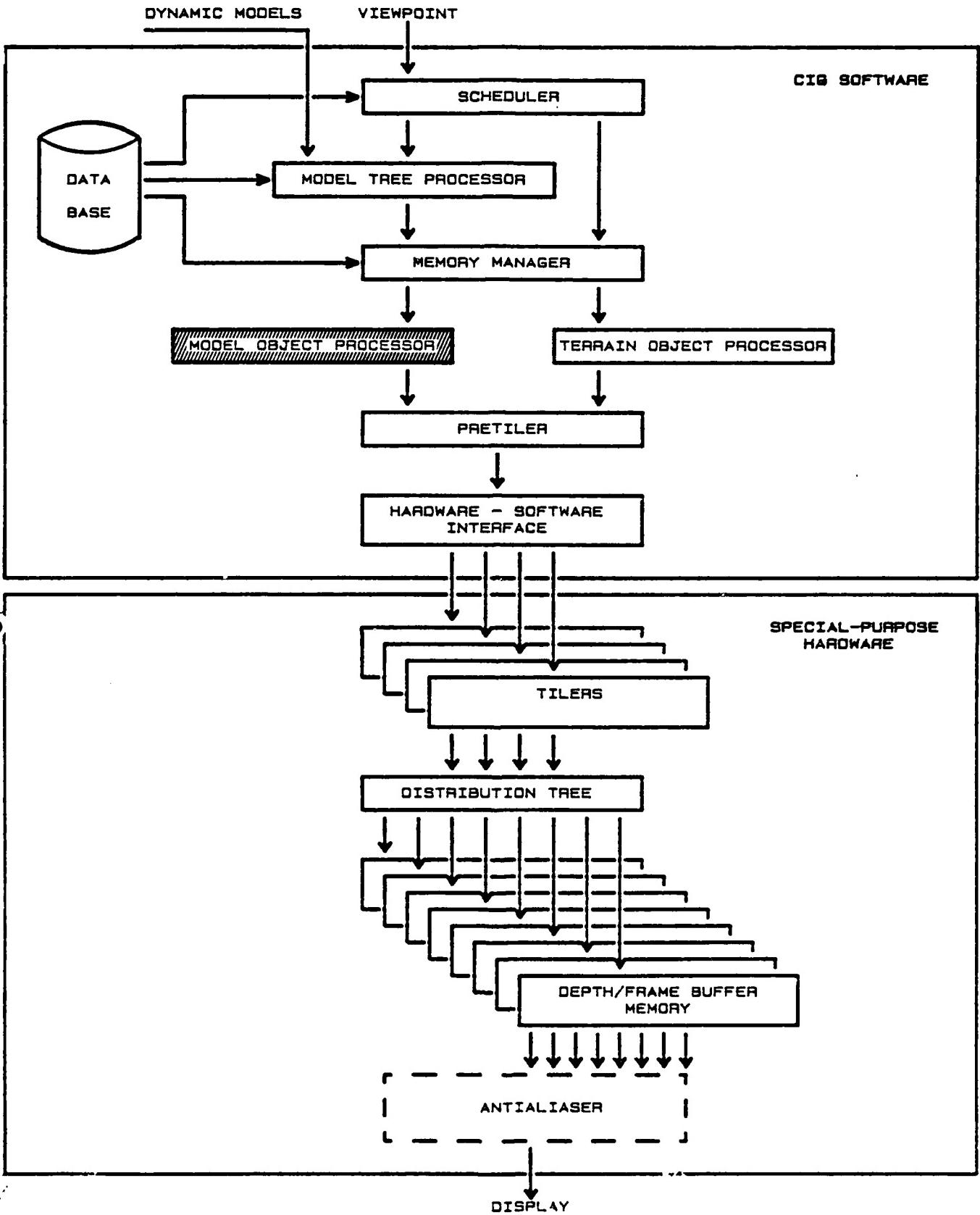
Figure 57. Clipping to Screen Space

These $(n-2)$ triangles are coplanar and share the same color and depth gradients. However, since these triangles are projected into 2-space as separate entities, this information is lost and the color and depth gradients are redundantly computed for each triangle. By clipping in 2-space, color and depth gradients are calculated once for each original projected triangle and this information can then be associated with all triangles created from the original. Therefore, clipping in 2-space allows the pretiler to avoid unnecessary calculations.

Once all clipping has been performed, the resultant n -gon is triangulated for processing by the tiler. The tiler has two requirements. The triangle vertices must be sorted on i and the corner vertex of the triangle must be defined. (For an explanation, see Section 5.2 and Figure 52.) At this point, the vertices, base color and depth, and color and depth slopes for each of the new triangles are sent to the hardware-software interface.

The calculations for projected color are currently not correctly implemented in the hardware version of the tiler, since the color calculations use a linear dependence of color on i and j (see Section 4.2 for an in-depth explanation). The color calculations are correctly implemented in software versions of the tiler and pretiler.

Future improvements will probably center on improved clipping strategies. Presently a design whereby the tiler performs the screen clipping is being tested. This completely obviates the need for clipping in the pretiler. Should this prove impractical, another improvement might be to modify the pretiler to process n -sided polygons. The pretiler might clip to all planes in viewpoint space (using Sutherland/Hodgman algorithm (Suth74a) or the Blinn/Newell algorithm (Blin77a)) to form an n -sided polygon. Projecting this n -gon to screen space, the pretiler would calculate the color and depth slopes, retriangulate, format, and send the appropriate data to the tiler.



MODEL OBJECT PROCESSOR
137

5.4 MODEL OBJECT PROCESSOR (MOP)

5.4.1 MODEL OBJECT PROCESSOR OVERVIEW

The model object processor generates a list of triangle primitives that define a 3-D model for the generic triangle pretiler/tiler. The MOP converts a model-space description for the object (such as a building or truck) into viewing space coordinates, discards all back faces, and computes the shading of each vertex or face based on the illumination angle. This information is bundled on a per triangle basis and passed to the pretiler.

The model object processor is designed to handle the data from both static and dynamic models. A static model is an object that does not move, such as buildings, roads, and airports. Dynamic models are objects that have motion associated with them. This motion may be relative to another part of the model, or it may be relative to the rest of the world.

The model object processor accepts a concatenation of coordinate transformation matrices that transform the part from model space to world space, and then to viewpoint space. The concatenation may also include a matrix to define motion for a part or the entire object. Once the matrix has been defined, the object processor does not need to know where a part came from, whether it is a dynamic or static model, or even whether each part is from the same model. Only the concatenation of the matrices in the model tree processor (Section 5.7) determines what kind of object the part came from. This implies that multiple concurrent model processors can be used in a system with a large number of static or dynamic models.

5.4.2 PROCESSING REQUIREMENTS

The task of this processor is to convert data defining a three-dimensional object in an arbitrary ABC object space into a list of front-facing triangles in the UVW viewpoint space. The coloring of the UVW triangles must also be determined by this processor given the sun vector.

To perform these operations, the MOP uses system inputs (such as viewpoint) and a model data definition that uses pointers to take advantage of shared vertex

information. There are several flags in the data definition that allow the model to be shaded in various manners. Depending on the shading technique used (Gour71, Bui75a, Blin77a, Blin77b, Duff79), varying amounts of data compaction may be realized.

5.4.3 PROCESSOR INPUTS

System information which is dependent upon the given frame being computed, and data definition which is the frame-invariant data that defines the model are the two types of input that are provided to the processor.

5.4.3.1 SYSTEM INFORMATION

System information is composed of the following items.

- A. The XYZ world-space viewpoint.
- B. The XYZ world-space sun vector.
- C. A 4x3 transformation matrix that transforms XYZ world-space to the UVW viewpoint space.
- D. A 4x3 transformation matrix that transforms the vertices of the model from its ABC model space into the XYZ world space.

Note that the transformation matrices are actually 4x4 matrices where the right-most column is assumed to be (0,0,0,1).

5.4.3.2 DATA DEFINITION

The data definition inputs are described below:

- A. A record containing information about the model data:
 - 1. Number of triangles in the model.
 - 2. Number of vertices in the model.
 - 3. Two flags:
 - Shading (face shading or curve shading).
 - Coloring (polygon colors or vertex colors).

- B. A list of triangles, each containing three vertex pointers.
- C. A list of vertices, each containing three 4-byte real numbers that specify the vertex's location in ABC space.
- D. A list of (red,green,blue) colors, which correspond to the entries in the triangle or vertex lists depending upon the setting of the coloring flag.
- E. A list of vertex normals, which is only present if the shading flag is set to curve shading. Each vertex normal is a ABC space vector that defines a normal to the surface. In order to reduce the amount of memory required for these triplets, each normal contains three 2-byte (16-bit) integers. These values are converted to 32-bit floating-point values within the algorithm.

5.4.3.3 DATA DEFINITION OPTIONS

The two flags that define the shading and coloring of the model affect the manner in which the data is processed. Depending upon the setting of these two flags, four different paths are taken within the processor: curve-shaded and polygon-colored; curve-shaded and vertex-colored; face-shaded and polygon-colored; and face-shaded and vertex-colored. The shading and coloring options are described below.

5.4.3.3.1 SHADING

A polygon can either be curve-shaded or face-shaded. When vertex normals are defined for the model, an intensity can be computed for each vertex by dotting the sun vector with the vertex normal. This is referred to as curve shading, since the resulting polygon will appear to be curved rather than planar. When the surface normal perpendicular to the plane defined by the vertices is dotted with the sun vector, a single intensity is computed that is applied to all of the vertices of the polygon. This results in a polygon that appears planar and is referred to as face shading.

If all vertex normals for each polygon in the model are equal to their surface normals, the model could have been defined as either curve-shaded or face-shaded (Bui75b). It is therefore somewhat misleading to think of a model with vertex normals to be a curve-shaded model. In the situation above, the model will still appear face-shaded; it is just that the data were defined as if it was a curve-shaded object.

5.4.3.3.2 COLORING

Color is defined using an RGB color space where each element (red, green, and blue) is an integer ranging from 0 to 255. The color of a polygon can either be defined by associating a RGB triplet with each vertex of the polygon, or by associating a single RGB triplet to the entire polygon. When RGB triplets are defined for each vertex, the process is called vertex coloring. When they are defined for each polygon, the result is referred to as polygon coloring.

As with the shading option, the choice between vertex and polygon information is not always clear. If each polygon has the same RGB triplet assigned to all its vertices, then the model can be considered either vertex-colored or polygon-colored. In reality, a model defined with this coloring scheme would be polygon-colored.

5.4.3.4 VERTEX DEFINITION

Now that the shading and coloring options have been defined, the effect these options exert on the model data can be investigated. Since a model is defined as a collection of vertices and a polygon consists of pointers to vertices, the vertex is the primitive building block of the model. The shading and coloring options affect the definition of the vertex by altering the amount of information that is stored at each vertex, as indicated in Table 12.

Table 12. Shading and Coloring Options

shading	colors	vertex definition
curved	vertex	x, y, z, xnorm, ynorm, znorm, red, grn, blu
curved	polygon	x, y, z, xnorm, ynorm, znorm
face	vertex	x, y, z, red, grn, blu
face	polygon	x, y, z

Different models achieve different levels of data compaction depending on the proportions of vertex and face data. Software was developed to represent a model in its most compact form.

5.4.4 INTERNAL MEMORIES

The processor incorporates an internal UVW memory which is used to eliminate redundant calculations of the UVW coordinates. A pointer from the ABC space vertex memory to the UVW memory is created whenever a vertex is transformed from ABC space to UVW space. When a triangle is processed, the pointers of the vertices are checked to determine if transformation has already been performed for those points.

Similarly, an internal RGB memory (or intensity memory) is used to eliminate redundant calculations of the RGB vertex colors (or the vertex intensities).

5.4.5 ALGORITHM

The MOP eliminates back-facing polygons, transforms the vertices to viewpoint space, and computes the vertex colorations. An outline of the algorithm used by the processor to perform this task is shown in Table 13.

**Table 13. Model Object Processor Algorithm
(Continued on the following page)**

Clear the internal UVW vertex memory (set pointers to nil).

Transform XYZ viewpoint and XYZ sun vector to ABC model space.

1. Initialize variables required for inverse transformation.
2. Transform the XYZ viewpoint to ABC model space.
3. Transform the XYZ sun vector to ABC model space.
4. Normalize the ABC model space sun vector.

Concatenate the XYZ - UVW transformation matrix to model's transformation matrix.

For each polygon do
begin (* polygon loop *)

Table 13. Model Object Processor Algorithm
(Continued on the following page)

Back face elimination in ABC model space.

1. Generate the ABC model space face normal.
 - a. Compute vectors from vertices.
 - b. Generate face normal by cross product.
2. Generate view vector from model viewpoint.
3. Perform the dot product back face test.

if front facing polygon then

begin (* front facing polygon *)

If face-shaded, compute polygon intensity of polygon RGB.

1. Normalize face normal.
2. Dot face normal with ABC model space sun vector.
3. Compute intensity of polygon.
4. If polygon-colored, compute final RGB color.

Transform vertices to UVW space.

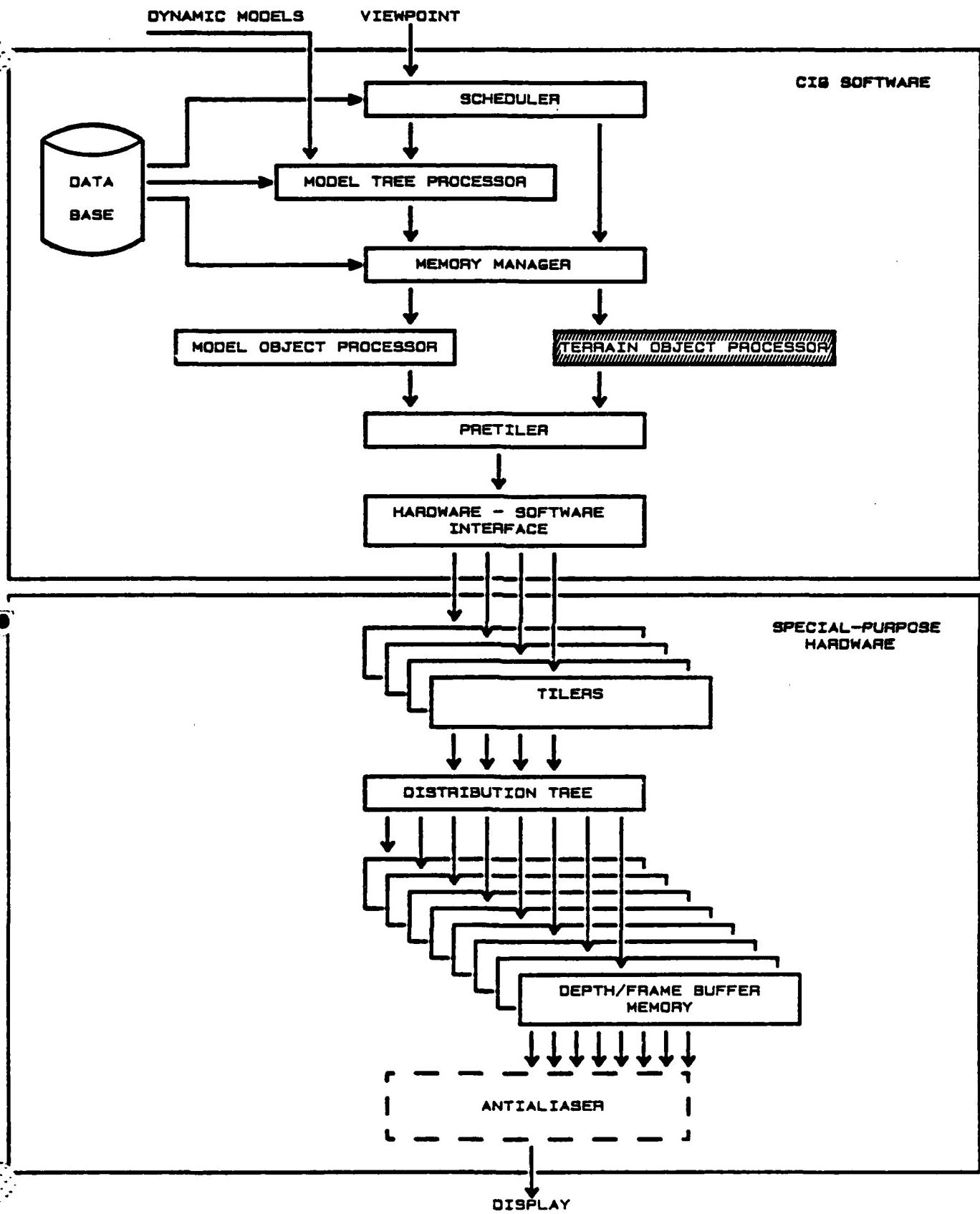
1. Clear UVW vertex memory if transformed vertices will not fit into the UVW memory.
2. Check if vertex is in the UVW vertex memory.
3. Transform vertex to UVW and store in vertex memory.
4. If curve-shaded, compute vertex intensity or RGB.
 - a. Dot ABC vertex normal with ABC model space sun vector.
 - b. Compute normalized intensity of vertex.
 - c. If vertex-colored, compute final RGB color and store it in the UVW vertex memory.
 - d. If polygon-colored, store vertex intensity in the UVW vertex memory.
5. Recall UVW and address of UVW vertex from the UVW vertex memory and store in output polygon.

Table 13. Model Object Processor Algorithm
(Concluded)

```
Compute final RGB color of vertices.  
1. If curve-shaded and vertex-colored, recall final RGB color  
   from vertex memory.  
2. If curve-shaded and polygon-colored, compute final RGB color  
   with vertex intensity from vertex memory.  
3. If face-shaded and vertex-colored, compute final RGB color  
   with polygon intensity.  
  
Send polygon to pretiler.  
  
end; (* front facing polygon *)  
end; (* polygon loop *)
```

5.4.6 Alternatives

Alternative data-type representations will require redesigning of the MOP. For example, if a model were represented with curved surfaces, little of the current processor would be applicable. In fact, a rudimentary quadric surface MOP was designed and implemented; however, further use of quadric surfaces would require extensive research to find the optimal set and sequence of processing steps.



TERRAIN OBJECT PROCESSOR
145

5.5 TERRAIN OBJECT PROCESSOR (TOP)

5.5.1 DESIGN CRITERIA

The objective of the terrain object processor is to represent topography with high realism and reduce the computational load on the system while doing so. Defense Mapping Agency (DMA) data is used to achieve a high degree of accuracy in terrain modeling and can be supplemented with topographic maps or other sources. Since the depth-buffer solution to the occlusion problem negates the need for separation planes and clusters, terrain databases are quickly generated in a totally automated process. The DMA data structure is a regular grid of elevation points that form a square mesh of topographic data. This structure is preserved and exploited by the TOP. Two alternatives to the grid structure involve 3-D modeling of the terrain or fitting the topography to a mathematical function that describes the terrain.

Evans and Sutherland, Singer Link, General Electric, and others currently use the 3-D approach to terrain modeling in their visual systems. The topography in these systems is rather simplistic in order to reduce the polygon count and to keep the cost of modeling (involving separation planes and clusters) reasonable (Scha81). The terrain database is meant to be suggestive of the topography and does not reflect actual elevation values, although some key features may be highlighted. Since these systems use separation planes between objects to help solve the hidden-surface problem, databases can become very complex, take a long time to construct, and require an inordinate amount of computation for creation.

Since the DARPA system uses the depth-buffer approach instead of a list-priority scheme, the drawbacks to 3-D modeling using separation planes disappear. The size of the database is, however, still an issue. The ability to model unusual areas of the terrain as irregular 3-D models is still considered advantageous and allowed as a special case of triangulated irregular networks (TIN) in the DARPA system.

Grumman Aerospace Corporation is developing a system that defines the topography as a collection of second-order curves with generic texture functions for scene detail (Stei83). By using second-order curves to describe the terrain, the number of separation planes and the amount of data required to form a database is reduced significantly. The main problem with using functions to generate the

topography is that major features are fairly well modeled, but secondary features are fitted statistically. Statistical representations do not necessarily correspond to the actual terrain and are not acceptable in situations where the mission has random movement requiring an accurate representation of all major and secondary terrain components. Rendering of functions is performed basically by a ray-tracing algorithm that finds the intersection of the line of sight through a pixel to the curve. Ray tracing provides good results, but has a high computational overhead (Fole82). Although this technique was not chosen for rendering terrain, it should be noted that it could be used in parallel with other processors currently used by the DARPA system in a situation requiring extreme compaction of databases. The DARPA system can also apply texture to the terrain surface by using the mip map technique discussed in Section 6.4. The texture mapped to the terrain is not a texture function, and can be either specific detail of an area of interest or generic texture such as wheat fields.

5.5.2 OVERVIEW

The terrain object processor (TOP) prepares the terrain geometry for processing by the tilers. The processor performs the back-face test, transforms the vertices from world space to viewing space, and assigns illumination values to the vertices of the triangle primitives that are the output of the TOP. The same general steps are taken by the model processor. However, the terrain has a regular data structure that lends itself to computational optimization.

Terrain objects and models differ in that terrain is described as a regular grid of elevation differences, and models are described as an irregular connection of data points. By taking advantage of the regular structure of terrain data, the TOP reduces the operations for the computation of the face normal from three multiplications and two additions per face to two additions for every two faces. Similarly, the operations for the transformation of terrain vertices from world space (x,y,z) to viewing space (u,v,w) is reduced from nine multiplications and nine additions per vertex to three multiplications and six additions.

Another advantage of the grid structure is that the terrain geometry is stored as a two-dimensional array of elevation differences with x,y coordinates implied from the location of the terrain "patch" in the database. By storing only the elevation

data and no x,y data, the amount of description geometry is approximately one third of the amount needed for models.

There are five distinct steps in the processing of terrain (see Figure 58):

- A. Reset the i,j (screen space) buffer and the intensity buffer.
- B. Compute the proper level of detail (LOD) for all terrain objects.
- C. Transform the geometry from world space (x,y,z) to view space (u,v,w).
- D. Determine which faces are front or back facing.
- E. Compute the proper lighting intensity from the illumination source(s) and the terrain geometry.

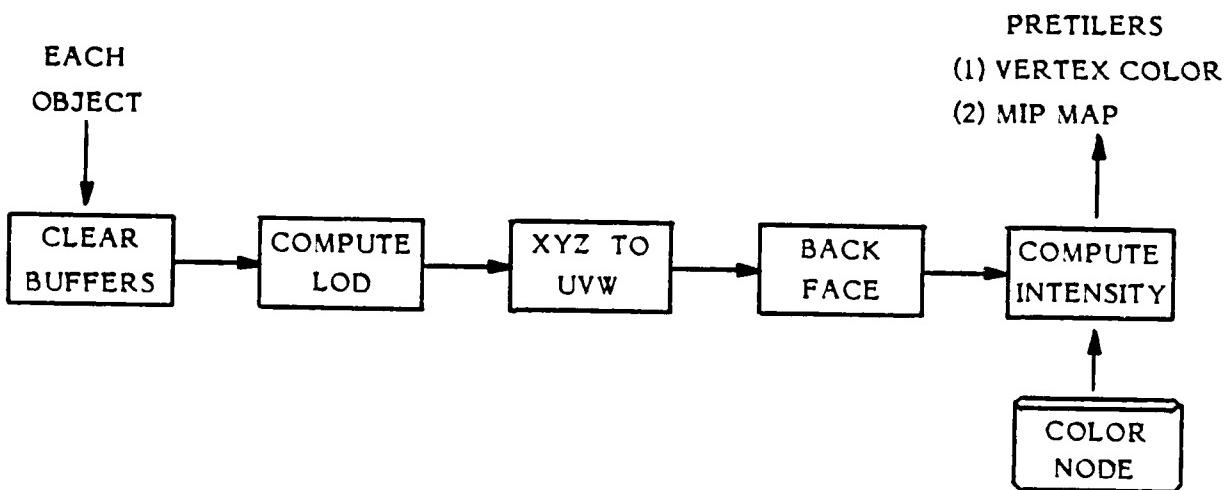


Figure 58. Terrain Object Processor Functional Flow

Upon completion of the intensity calculation, the vertices describing a terrain face are passed to the pretiler for conversion to screen space coordinates and rasterizing. The following sections will describe the terrain data structures and processing in greater detail.

5.5.3 TERRAIN DATA STRUCTURES

Two types of data structures are used to represent terrain cover:

- A. Regular grids.
- B. Triangulated irregular networks (TIN).

The regular grid is the easiest to use and can be formed from many different sources. TIN's are useful in defining radically shaped terrain cover, but must be processed in a more general manner.

5.5.3.1 REGULAR GRIDS

Most of the regular grids are formed from Defense Mapping Agency (DMA) Digital Terrain Elevation Data (DTED). However, databases can be built from terrain model boards, topographic maps, and data derived from stereo photography. The terrain data is stored as an array of elevation data, followed by an array of vertex normals.

The first element in the elevation array is the absolute elevation value (to the nearest meter) of the lower left corner of the terrain region (see Figure 59). The remaining vertices are offsets from each preceding vertex. Each delta z in the first column is the difference in elevation from the vertex below it. This means that the elevation of vertex N in the left-most column can be computed by summing the previous N values. The remaining vertices contain the difference in elevation from the vertex immediately to its left. Any vertex N,M can be computed by summing the first N elevation values of the left-most column and then summing the next M-1 delta-z values across the row.

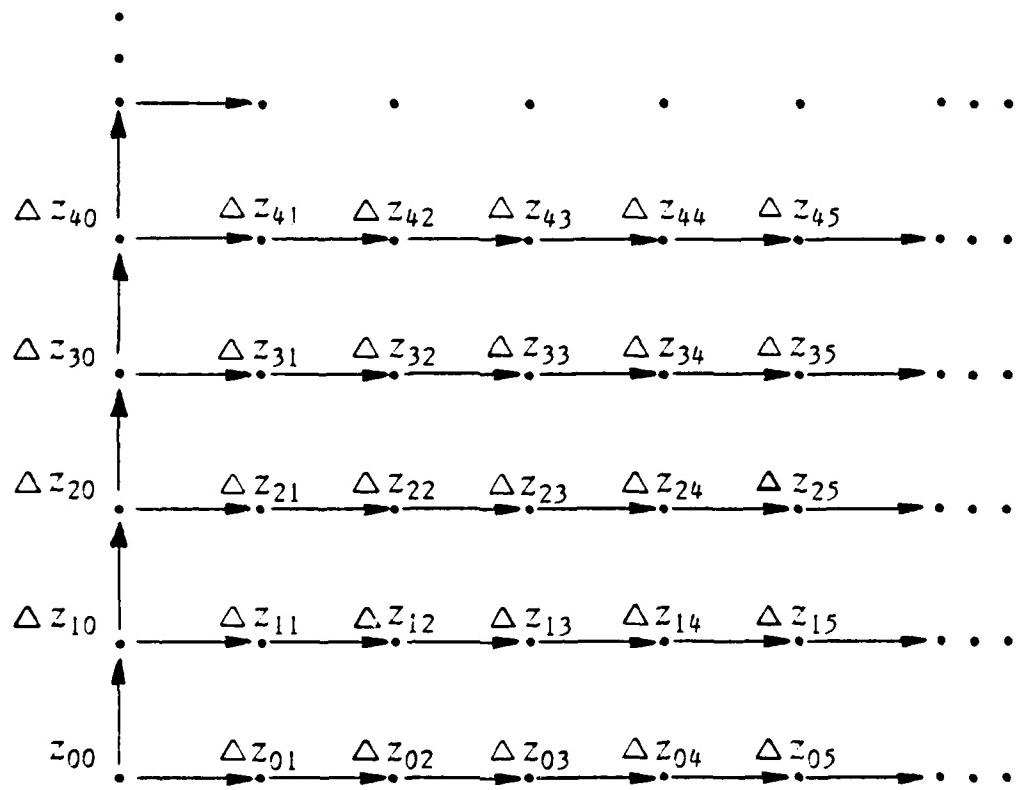


Figure 59. Terrain Delta-Z Grid

Although this method of data representation appears cumbersome at first glance, it greatly reduces the amount of processing necessary for the back face test and for the transformation of vertices from world space to viewing space. Figure 60 illustrates the DMA terrain grid structure.

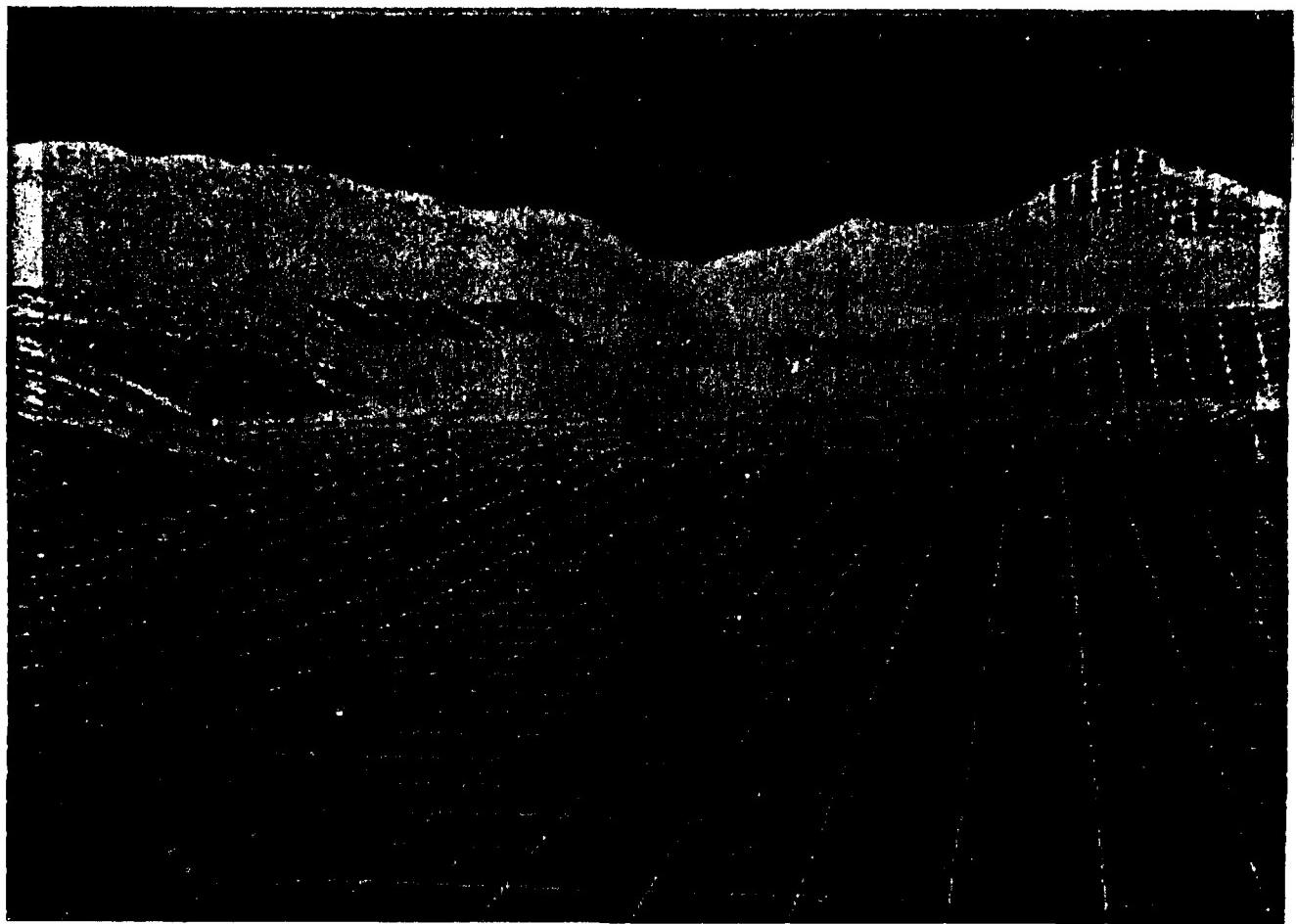


Figure 60. DMA Terrain Grid

5.5.3.2 TRIANGULATED IRREGULAR NETWORKS (TIN)

A triangulated irregular network (TIN) is simply a random grouping of coordinates that is sufficient to represent a feature, and may then be triangulated to form a model of that feature (Mita82). TIN's are useful in describing areas of the world that have abrupt features in an otherwise regular topography. An example could be a small creek meandering through a relatively flat plain.

The TIN helps to solve two problems in this case. First, the creek may be small enough that to properly define it with a regular grid would require an unreasonably number of sample points. A TIN, however, can define the feature as it actually exists without any rigid structure and a minimum of vertices. Secondly, the

surrounding plain does not require many sample points to describe it. In this case several data points can replace an otherwise redundant grid. By using a TIN in this case, the amount of data necessary to portray the topography is reduced, and the terrain is described more accurately than with a regular grid.

Since TIN's are highly irregular, they fall more reasonably into the data description of a model and are processed through the model processor. The remainder of this section on terrain processing refers only to the processing of regular grids.

5.5.4 SHARED VERTEX MEMORIES

Since the basic primitive of the tiler architecture is a triangle, many vertices of both models and terrain grids are shared. In the case of terrain, a single vertex can be shared by six triangles. In order to take advantage of this fact, the terrain object processor has a memory in which the intensity (or color) is stored for each vertex within a terrain object. Before the vertex illumination is computed, the memory is checked to see if the vertex has been previously processed. The pretiler also takes advantage of this fact and queries its i,j screen-space buffer so that multiple projections from view space to screen space are avoided. Statistics taken from the DARPA system have shown that in the case of terrain processing, 70% of the triangle vertices can be found in the shared-vertex memories during a given frame. Thus, the shared vertex memories greatly reduce the amount of computation required to generate a scene.

5.5.5 LEVEL OF DETAIL

An important measure of system throughput is the number of faces (polygons) that are processed in a given frame. This number can be meaningless if most of the faces are far off in the distance and are projected to a few pixels on the screen. This loads down the system and adds little or no complexity to the scene. Level-of-detail control for terrain objects is essential for improved system throughput and image complexity. The thrust of the terrain level-of-detail effort was to keep the screen coverage of each face constant. This implies that those faces farther from the viewpoint must be proportionately larger than the faces in the foreground. A LOD control algorithm used in the terrain object processor modifies the

size of the terrain vertex spacing, based on the distance from the viewpoint. A complete explanation of the LOD control algorithm and research is in Section 6.5.

5.5.6 COORDINATE TRANSFORMATIONS

Before the vertex data can be passed to the pretiler, the x,y,z world coordinates must be transformed to the u,v,w viewing coordinates. This is traditionally done by multiplying the world coordinate vertex with the viewing transformation matrix which yields the new coordinates. The transformation matrix is a concatenation of the translation, yaw, pitch, and roll values that define the position, direction, and orientation of the viewpoint.

$$(u, v, w) = (x, y, z, 1) \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ t_1 & t_2 & t_3 \end{bmatrix} \quad \text{Eq. 1}$$

where $u = x a_{11} + y a_{21} + z a_{31} + t_1$
 $v = x a_{12} + y a_{22} + z a_{32} + t_2$
 $w = x a_{13} + y a_{23} + z a_{33} + t_3$ Eq. 2

The previous transformation requires nine multiplications and nine additions to transform each vertex. The computation for each vertex can be radically reduced by using a delta-z grid of regularly-spaced data points and summing the values to obtain the transformed points.

The grid summation method is based on the fact that each successive vertex transformation is merely the previously transformed coordinate plus the change in x (or y) and z. Since the change in both the x (d_x) and y (d_y) directions is a constant, the u,v,w components of this change are calculated once before the beginning of the grid transformation:

$$\begin{aligned}Dx_u &= d_x a_{11} \\Dx_v &= d_x a_{12} \\Dx_w &= d_x a_{13}\end{aligned}$$

and

Eq. 3

$$\begin{aligned}Dy_u &= d_y a_{21} \\Dy_v &= d_y a_{22} \\Dy_w &= d_y a_{23}\end{aligned}$$

The first world space coordinate in the grid must be transformed to view coordinates using the standard-transformation matrix multiplication method. This gives us the seed u, v, w coordinate on which the rest of the grid is based at a setup cost of fifteen multiplications and nine additions.

From this point, all remaining vertices are transformed as a series of additions with one multiplication per coordinate. The next coordinates in the x direction are merely:

$$\begin{aligned}u_1 &= u_0 + Dx_u + Dz a_{31} \\v_1 &= v_0 + Dx_v + Dz a_{32} \\w_1 &= w_0 + Dx_w + Dz a_{33}\end{aligned}\quad \text{Eq. 4}$$

These coordinates are derived from the difference in the coordinate transformation equations for the first point (P_0) and the equation for the second point (P_1).

$$\begin{aligned}P_0: \quad u &= x a_{11} + y a_{21} + z a_{31} + t_1 \\P_1: \quad u' &= (x + d_x) a_{11} + y a_{21} + (z + Dz) a_{31} + t_1\end{aligned}$$

$$\begin{aligned}(\text{expand}) \quad u' &= x a_{11} + d_x a_{11} + y a_{21} + z a_{31} + Dz a_{31} + t_1 \\(\text{diff}) \quad u' - u &= d_x a_{11} + Dz a_{31}\end{aligned}$$

Since $d_x a_{11}$ is a constant (from equation 3), the equation for the new point P_1 becomes:

$$\begin{aligned} u' &= u + D_x u + D_z a_{31} \\ v' &= v + D_x v + D_z a_{32} \\ w' &= w + D_x w + D_z a_{33} \end{aligned}$$

Eq. 5

The coordinates in the y direction are similarly derived.

The savings per vertex in the grid are six multiplications and three additions less than with the standard matrix multiplication method (see Table 14).

Table 14. Per Vertex Coordinate Transformation Cost

	Multiplications	Additions
Matrix	9	9
Grid	3	6

5.5.7 BACK FACE ELIMINATION

In a polygon-based system, there are two types of polygons in any given scene: front-facing polygons, and those facing away from the viewer (back faces). The back-facing polygons are never visible in a continuous model and are hidden from view by front faces. For example, consider a box as seen from an arbitrary view (Figure 61).

The box, as seen from this view, has three front faces (front, right side, and top) and three back faces (left side, back, and bottom). Since back faces are never visible, it is desirable to remove these faces from processing at the earliest possible moment.

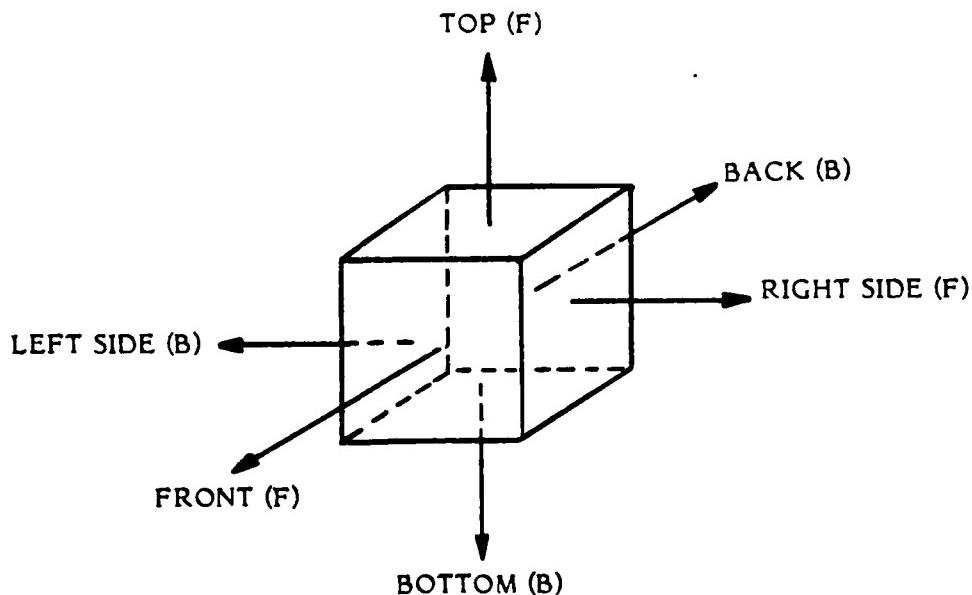


Figure 61. Six-sided Box with Three Front Faces and Three Back Faces

Statistics taken from the DARPA system have shown that for any given scene, between 30% and 50% of all terrain faces are back faces. To reduce the throughput on the system, a back-face elimination step is performed on all objects in the TOP.

There are two basic steps in solving the back-face test. First, the normal to the face is calculated to determine the face's orientation. This involves taking the cross product of two of the vectors forming the sides of the terrain face. Second, the dot product of the view vector (a vector from the viewpoint to a point on the face) and the normal vector is computed. A negative result from the dot product indicates that the face is directed toward the viewer, and a positive result indicates a back face. For an irregular grid of data points, the cross product for the normal calculation would take six multiplications and three additions per face.

$$\vec{A} \times \vec{B} = (A_x, A_y, A_z) \times (B_x, B_y, B_z)$$

$$(A_y B_z - A_z B_y, A_z B_x - A_x B_z, A_x B_y - A_y B_x) \quad \text{Eq. 6}$$

Because the terrain data is stored on a regular grid so that the x component of the vector A and the y component of vector B are both zero (see Figure 62), the cross product can be reduced to

$$\vec{A} \times \vec{B} = (A_y B_z, A_z B_x, -A_y B_x),$$

This simple fact reduces the cost of computing the cross product to three multiplications.

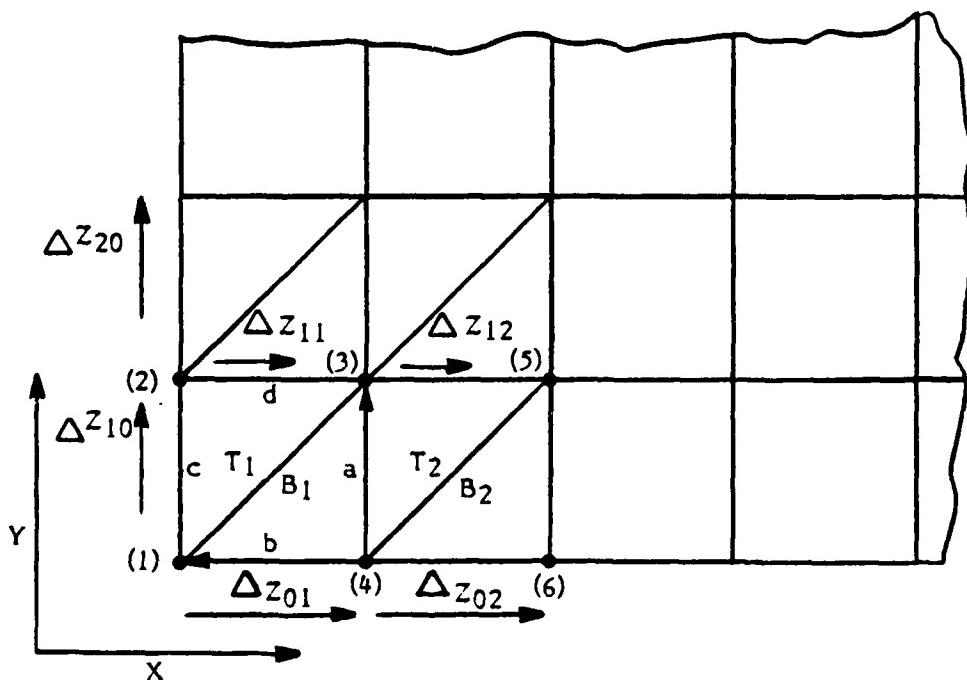


Figure 62. Terrain Face Construction

If the Digital Terrain Elevation Data (DTED) were not stored as elevation differences (delta z), an additional two additions would be necessary to determine the z components of the vectors used in the cross product.

If δx (d_x) and δy (d_y) are equal in the regularly-spaced grid, the cross product can be further reduced to

$$\vec{A} \times \vec{B} = (B_z, A_z, -B_x)$$

by removing a constant.

From Figure 62, it should be noted that the δz (d_z) information is stored for vertices from left to right in rows. The d_z for the interior columns can be computed for the bottom face as

$$A_z = C_z - B_z + D_z \quad \text{Eq. 6}$$

by way of the following derivation:

$$C_z = z_2 - z_1$$

$$B_z = z_4 - z_1$$

$$D_z = z_3 - z_2$$

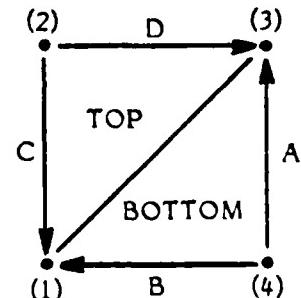
Eq. 7

$$A_z = (z_2 - z_1) - (z_4 - z_1) + (z_3 - z_2)$$

$$A_z = z_2 - z_1 - z_4 + z_1 + z_3 - z_2$$

$$A_z = z_3 - z_4$$

Eq. 8



This reduces the cross product computation to two additions

$$\vec{A} \times \vec{B} = (B_z, C_z - B_z + D_z, -B_x) \quad (\text{bottom face})$$

$$\vec{C} \times \vec{D} = (D_z, C_z, -D_y) \quad (\text{top face})$$

Eq. 9

Since C_z is the inverse of the A_z of the previous bottom face, the computation required to calculate the cross product for both faces is reduced to two additions. Table 15 shows the computational costs for the three methods of cross product determination.

Table 15. Cross Product Computation for Two Faces

	Multiplications	Additions
Standard	12	6
Grid	6	4
Delta Z	0	2

Once the normal to the face has been computed, the dot product of the normal and the viewpoint vector is performed. The sign of the dot product indicates whether the face is pointing away from the viewer ($\text{back} \geq 0$) or is facing the viewer ($\text{front} < 0$). Back faces are discarded at this time, and the amount of illumination is computed for all front faces.

5.5.8 ILLUMINATION COMPUTATION

The shading of the terrain objects assumes a diffuse reflector, and uses an algorithm based on Lambert's cosine law that includes both ambient and diffuse components of illumination. Lambert's law states that the intensity of the reflected light is proportional to the dot product (cosine of the angles) of the surface normal and the direction of the illumination. The ambient light is the proportion of the total illumination that is reflected to the surface from the atmosphere and scattering (Fole82).

The equation:

$$I = k_a + k_d(\vec{S} \cdot \vec{N})$$

where

k_a = ambient lighting constant

k_d = diffuse lighting constant

\vec{S} = incident light vector

\vec{N} = normal to the surface at a vertex

is used to compute the illumination intensity at each vertex of a terrain face. Before the computation is performed, the intensity buffer is checked to see if the intensity for that vertex has been computed previously. In the case of terrain, one vertex may be shared by as many as six faces.

The terrain may be shaded by one of two different methods. The first method is Gouraud shading. Gouraud shading uses the vertex normals to compute the individual intensities at each vertex of the face, and then linearly interpolates the intensity across the surface. This gives a smooth texture to the terrain and illuminates intensity discontinuities along edges. When Gouraud shading is used, the red, green, and blue components of color for each vertex are combined with the intensity of the vertex and passed to the pretiler. Figure 63 shows Gouraud shaded terrain of the Mount Rainier area in Washington State.

The other method for shading is texture mapping of a two-dimensional pattern, photograph, or texture over the surface. The method for applying the texture (mip map) is detailed in Section 6.4. If a texture map is used, the intensities of the vertices are bundled with the face data and passed to the texture tiler. Figure 64 shows texture mapping of the Mount Rainier area of Washington State.

5.5.9 FUTURE DEVELOPMENT

Terrain constitutes a major proportion of the data processed in nap-of-the-earth flight simulation. The presence of a specialized terrain object processor is therefore well justified by the savings in processing time resulting from optimized grid processing. However, certain CIG applications, such as computer-aided design or space simulation, may have no use for a TOP.

For those applications which do call for a TOP, its development will be primarily affected by changes in the LOD processing scheme. As described in Section 6.5, the current LOD algorithm calls for dynamic downsampling of the terrain grid coupled with a very simple method of processing LOD transitions. In the future, terrain objects of different levels of detail will be precomputed and stored at different levels of the database tree. This will simplify LOD processing, since the dynamic downsampling is no longer necessary. In addition, better LOD transition schemes will be developed.

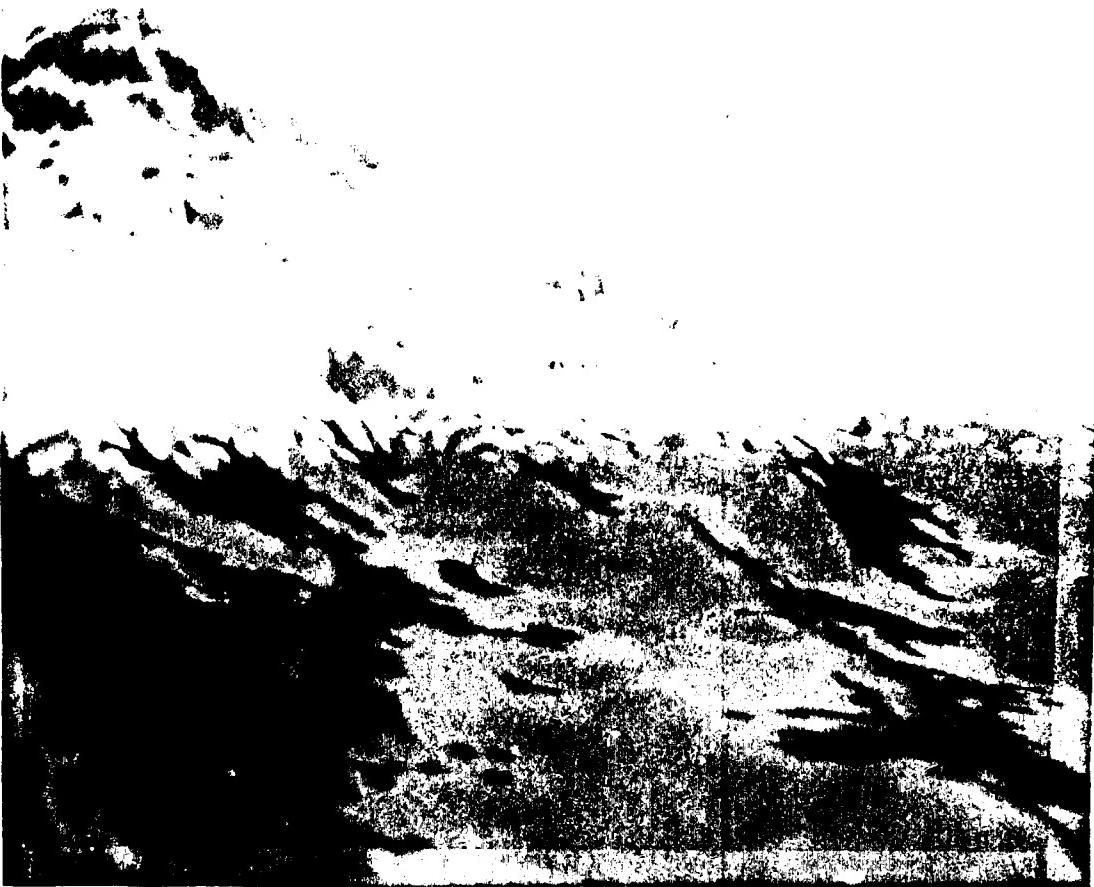


Figure 63. Mount Rainier - Geir and Shaded Terrain

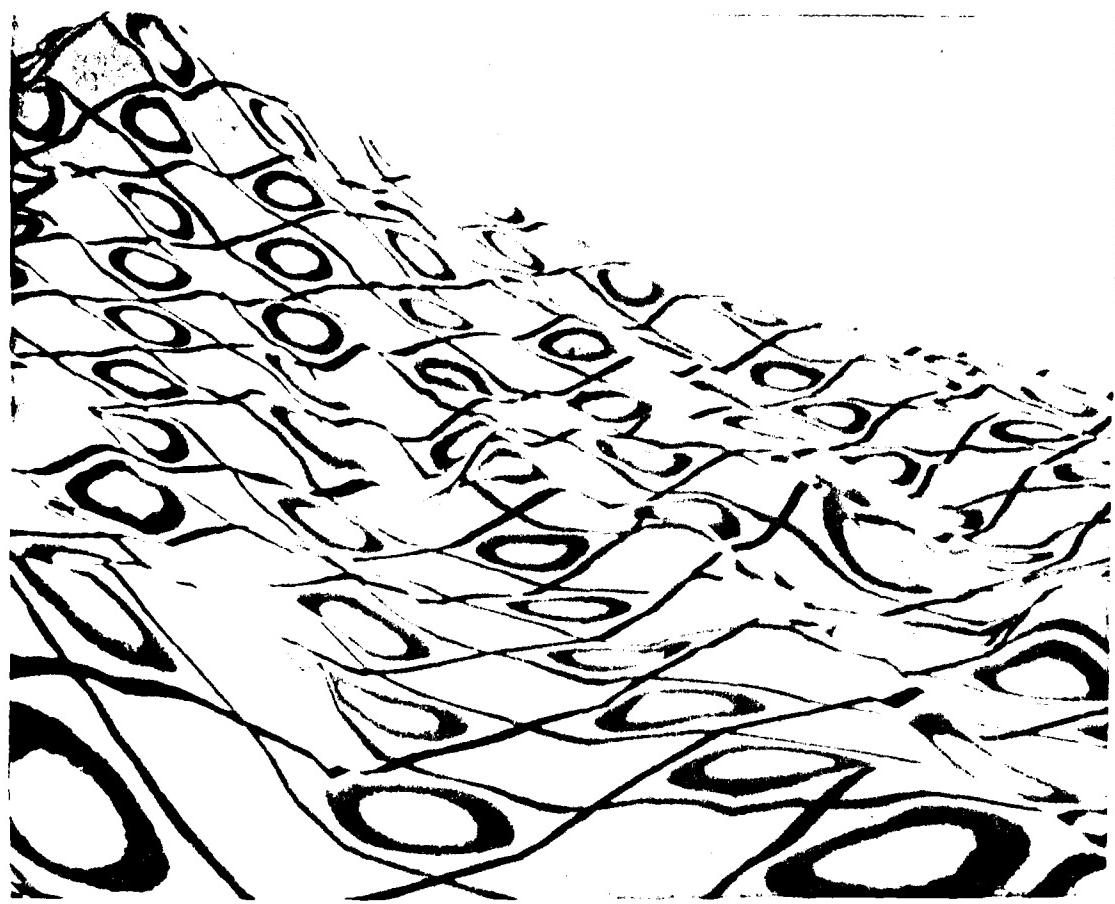
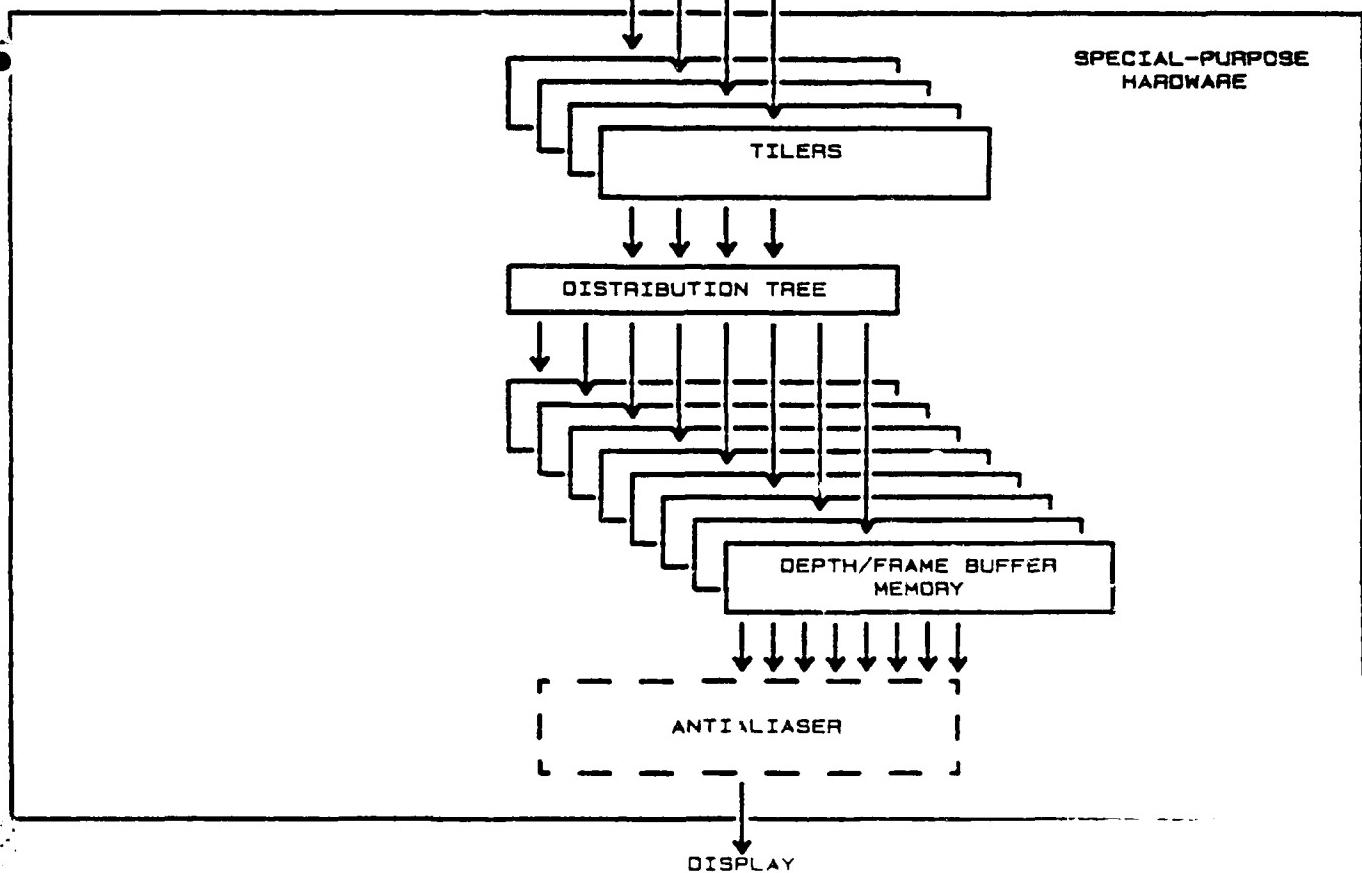
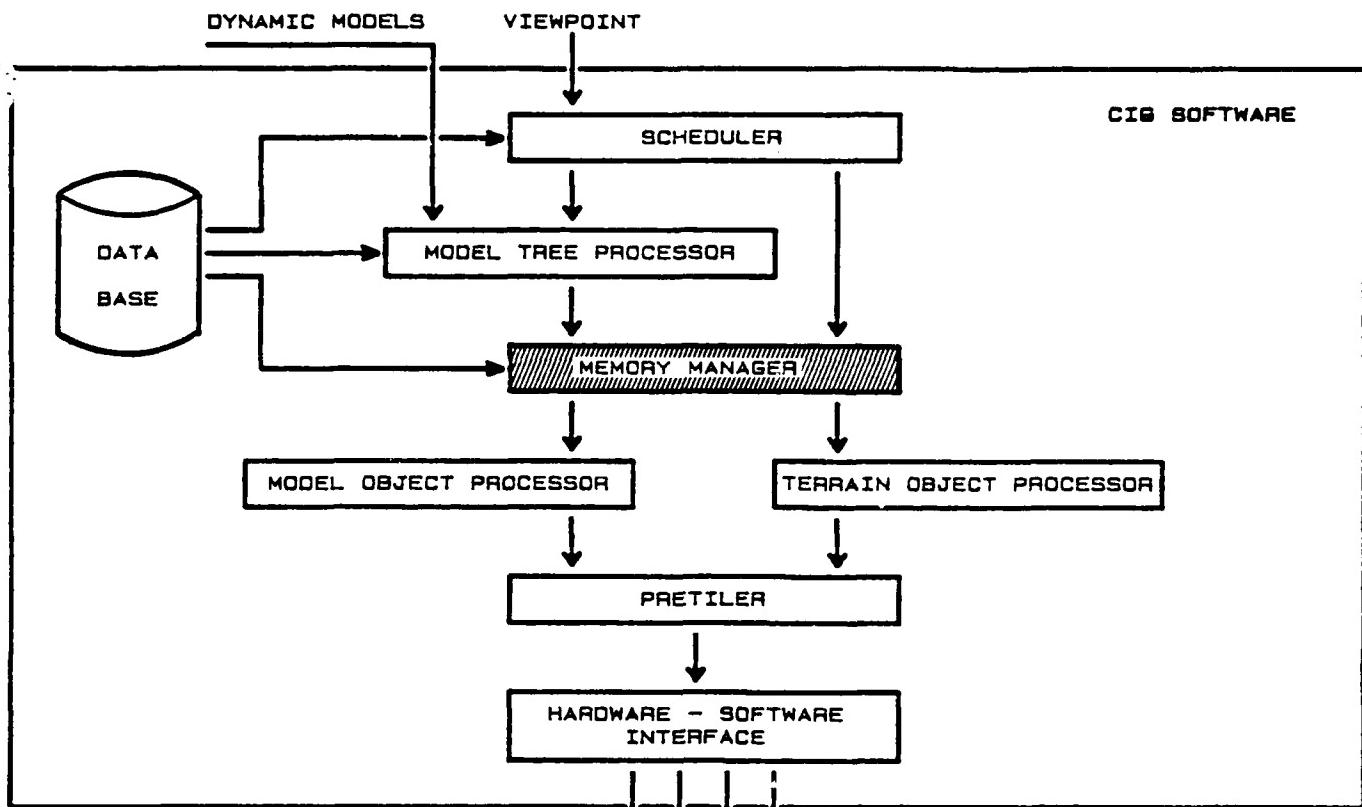


Figure 64. Mount Rainier - Texture Mapped Terrain

The major thrusts of future development will be a hardware implementation of the TOP, and a software system to build terrain databases from a wide variety of sources. Present and future database construction concepts are presented in Section 5.1.



MEMORY MANAGEMENT

RD-A141 083

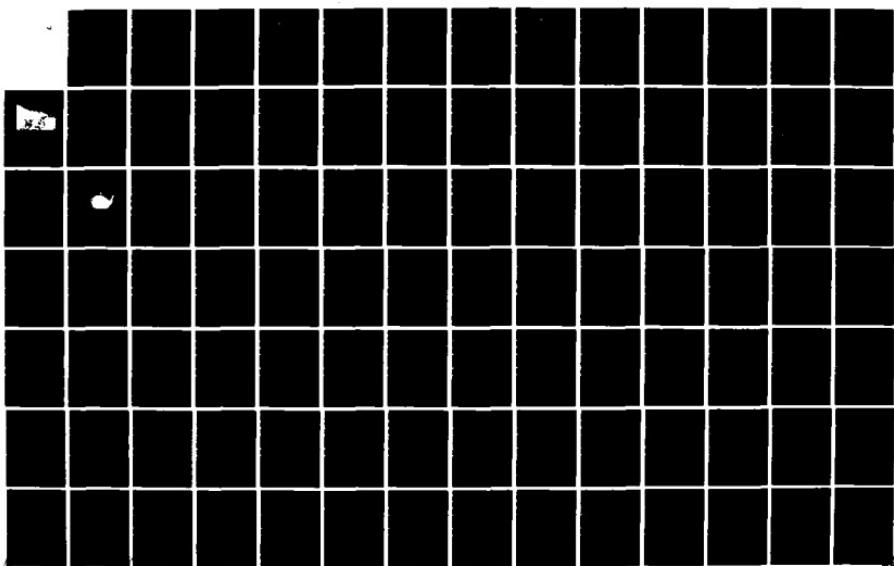
MULTIPROCESSOR Z-BUFFER ARCHITECTURE FOR HIGH-SPEED
HIGH COMPLEXITY COMPUTER IMAGE GENERATION(U) BOEING
AEROSPACE CO SEATTLE WA DEC 83 MDA903-82-C-0101

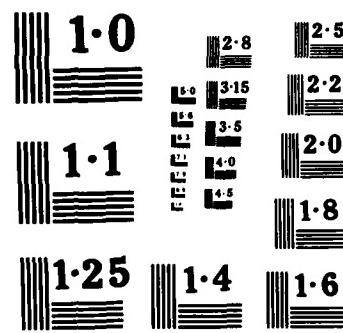
34

UNCLASSIFIED

F/G 9/2

NL





5.6 MEMORY MANAGER

The memory manager's main task is to perform all disk input/output (I/O) for the object processors. Frame-to-frame coherence and speed were the major considerations in the design. Coherence is achieved through internal list structures, and speed is realized by eliminating redundant and unnecessary disk I/O.

5.6.1 IMPLEMENTATION

Since disk I/O is by far the slowest operation involved in the image generation process, minimizing the number of disk accesses is the most effective way to improve the CIG system's speed. The manager is designed to read only necessary data and to keep it in memory as long as possible. A list is kept of all data currently residing in memory, and the manager processes resident data before overwriting any of it by reading non-resident data. This list is ordered by distance from the viewpoint, so that the most pertinent data is retained if there is insufficient memory to hold the entire list.

When the manager examines a node, one of three conditions exists:

- A. The node's data is resident. This is the simplest case. The manager need only send the starting address of the data to the appropriate object processor.
- B. The node's data is not resident, but there is enough free memory available to hold it. The manager then reads the data and sends the starting address to the appropriate object processor.
- C. The node's data is not resident, and there is insufficient memory available to hold it. In this case, the processing of the node is deferred until all nodes that fit either A or B have been processed.

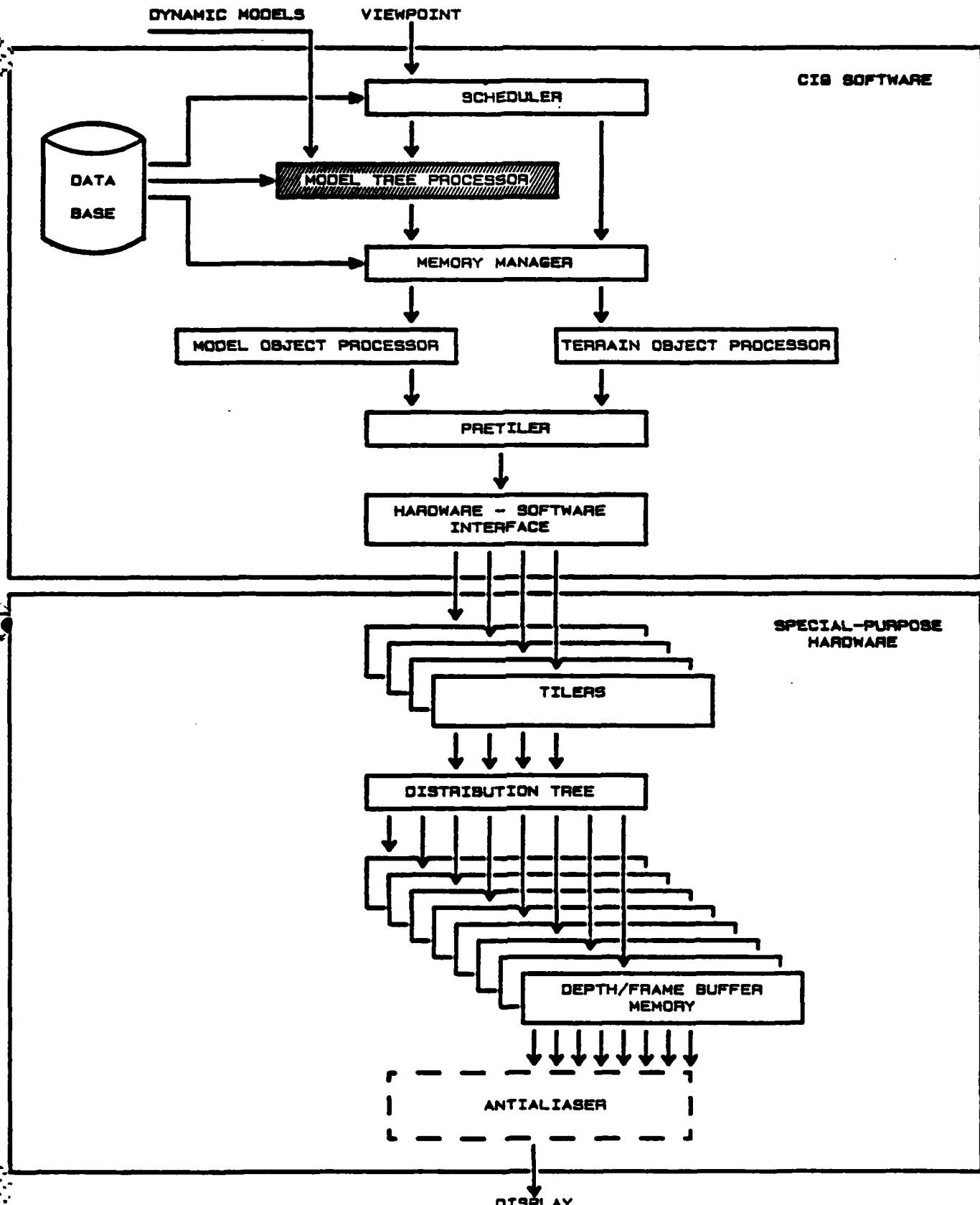
The existence of the memory manager not only increases speed, but also relieves the object processors of all I/O and memory management concerns. This leads to modularity of the overall design.

A good deal of the manager's work involves the manipulation of linked list structures. The ease with which Pascal handles these structures makes it a highly suitable language in which to express the algorithm.

5.6.2 FUTURE DEVELOPMENT

Several improvements in the memory management algorithm are needed to handle changes in the field-of-view set more efficiently. The FOV set is the set of nodes which are in the FOV. Currently, the FOV set change is computed by traversing the incoming node list, marking all data in memory which is needed, and freeing memory by releasing data which is not needed. The FOV set change is more logically and efficiently computed in the scheduler (see Section 5.8). A provision must also be made for nodes on the border of the FOV set; one must avoid deallocated data which may once again be in the FOV set of the next frame. The deallocation scheme therefore needs to be made "sticky".

In real-time systems, the high data rates between the memory manager and the object processors may dictate that the memory manager be placed in special-purpose hardware. Since the node lists are shared data structures between the scheduler and the memory manager, separation of the scheduler and memory manager may require changing their interface.



5.7 MODEL TREE PROCESSOR

5.7.1 DESIGN CRITERIA

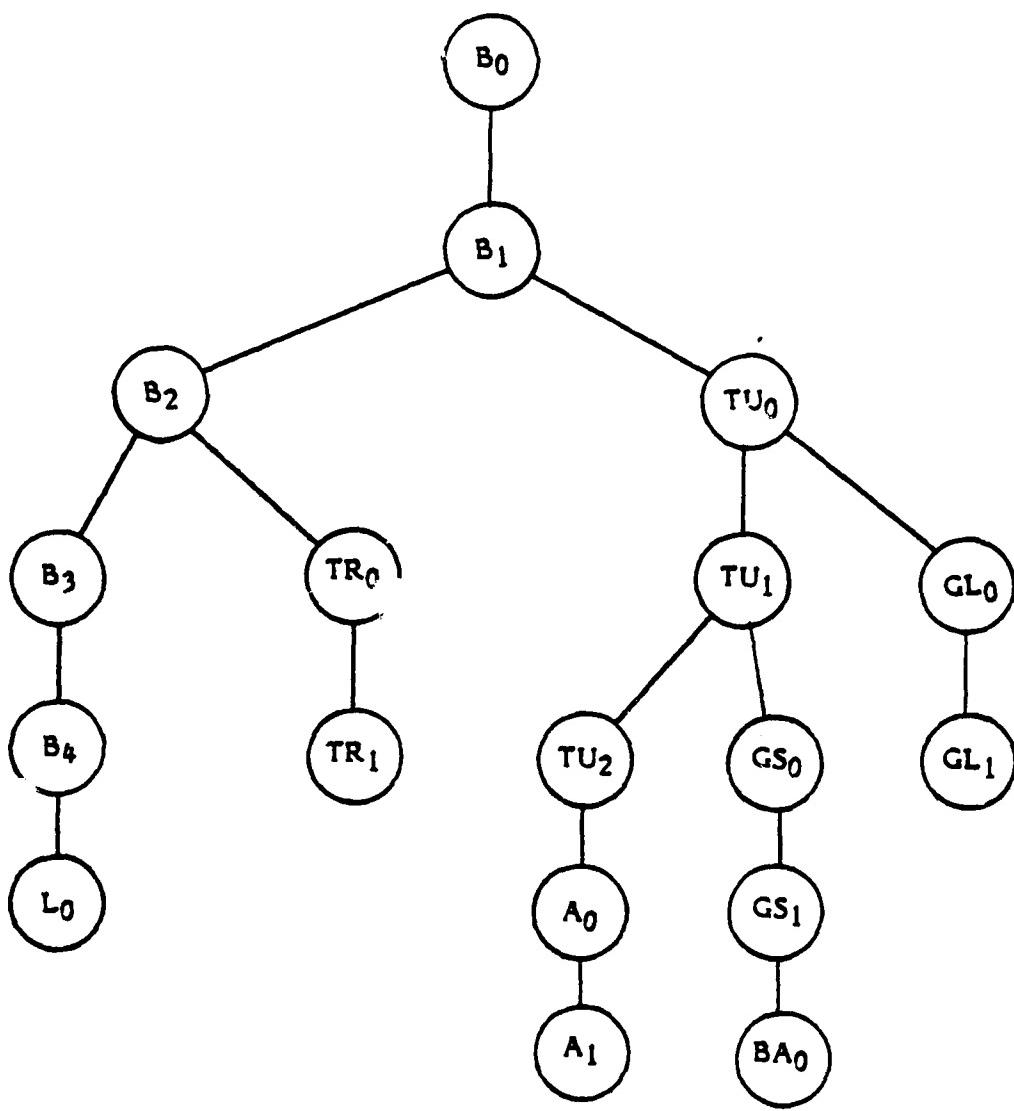
The major design criteria for the model tree processor (MTP) was that it be able to process a data structure which could describe both dynamic and static models. In addition, the data structure needed to be organized to permit easy traversal of model parts by FOV and LOD tests. The model tree embodies these design points and also allows for effective processing of moving and detachable parts.

5.7.2 PROCESSOR INPUTS

The MTP behaves like a second-level scheduler for models. As described in Section 5.8, the scheduler generates a list of static model nodes that are in the field of view. A list of dynamic model nodes is supplied from outside the scheduler to the MTP. Each model node points to a hierarchical model tree that is similar in structure to the gaming area database structure (Clar76, Rubi80, Keit81) (Section 5.1). A model tree consists of N parts that define a model at successively greater levels of complexity for LOD control. Information for the FOV test is also contained at each level of the tree to facilitate efficient FOV processing. Figure 65 describes the structure of the model object tree structure.

5.7.3 PROCESS DESCRIPTION

The model tree is traversed in a breadth-first order to determine if the model or part is in the FOV. If, at any level, the object is determined to be out of the field-of-view, the rest of the data in the tree is ignored. Similarly, once the object has been determined to be in the field of view, an LOD test is performed to determine the complexity of the parts to be generated (Rubi72). The data is ordered so that all parts on the same level of the tree are of the same effective level of detail, so only one LOD test is necessary for each level. The algorithm used is described in Table 16.



B_s - TANK BODY
 TU_s - TURRET
 TR_s - TREAD
 GL_s - GUN, LARGE
 GS_s - GUN, SMALL
 A_s - ANTENNA
 L_s - LIGHT
 BA_s - BARREL, AFT

SUBSCRIPT (s) INDICATES THE LEVEL OF DETAIL (LOD) FOR EACH PART.

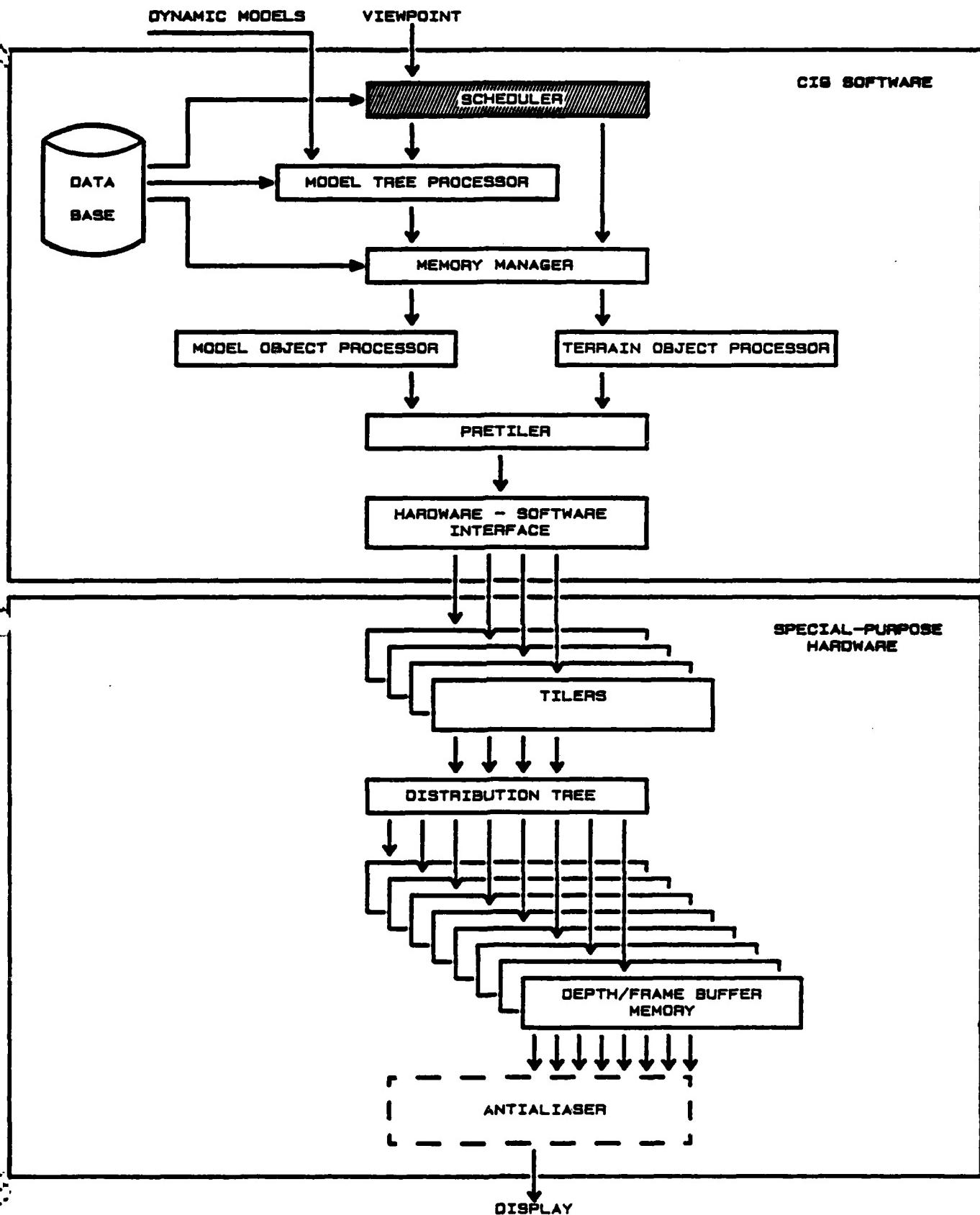
Figure 65. Model Object Tree Structure

Table 16. Model Tree Processor Algorithm

```
Place root node into queue;
While queue is not empty do begin
    Remove node Q from queue;
    If Q is listed in model table then begin
        Concatenate Q's transformation and default transformation;
        If Q contains data then begin
            If in_FOV then begin
                If correct_LOD then insert Q in output list;
                Place all Q's children into queue;
            end; (* FOV test *)
            end; (* Test for data *)
        end; (* Valid model test *)
    end; (* Node processing loop *)
```

5.7.4 PROCESSOR OUTPUT

The output of the MTP is composed of a list of nodes containing model parts and the transformation matrix necessary to transform the part from model space to world space. In the case of dynamic models, the MTP concatenates the matrix describing the motion of the part with the model-space-to-world-space transformation matrix. From this point on, there is absolutely no difference between static and dynamic models. The model node, containing a list of selected model parts, is then passed to the memory manager (Section 5.6).



5.8 THE SCHEDULER

The scheduler is a processor designed to efficiently traverse the database tree stored on disk. The design attempts to avoid restricting the flexibility of the database, and to traverse the tree with as little disk I/O as possible.

5.8.1 IMPLEMENTATION

The scheduler algorithm makes no assumptions about the structure of the database, and it preserves database flexibility by treating all nodes identically. The scheduler can therefore always determine the appropriate action from only the data contained within the current node.

Frame-to-frame coherence is maintained through several internal list structures. The free list contains all nodes that are not considered necessary in near-future frames. All nodes are put on the free list at initialization time, and the free list is thereafter treated as a node storage pool (Horo82). The want list is the destination for all nodes which represent no visible data but are suspected of being needed in near-future frames. The last list is collectively referred to as the need list, but actually consists of three lists: the TED list, color map list, and the model list, depending on the type of data it contains. These lists contain all nodes that are visible in the current frame, and will therefore probably be visible in the next frame.

At the end of computation for the current frame, the want list is appended to the need list, and the result becomes the new want list. As processing begins on the next frame, most of the nodes of visible objects are resident in memory on the want list, since the content of the frame does not vary greatly between two frames. When the scheduler traverses the tree, the nodes it needs can be retrieved from the want list by a simple relinkage, as opposed to a time-consuming read from disk. Note that this scheme does not require the entire tree to be in memory. Only the subset of the tree that is visible in each frame is in memory for that frame; only the relatively few nodes that enter the scene between two frames need to be read.

One attractive property of the tree structure is that the entire database is represented at each level of the tree; the difference between levels is only in the density of the information. The density is very high at the top of the tree, since the root node represents the entire database in one node. Moreover, if a node in the tree is not in the field-of-view, its children are also not visible. Therefore, when a node is found to lie outside the FOV, the entire subtree rooted at that node can be ignored. These two properties were recognized in the design of the scheduler, so that it disposes of nodes and, therefore, subtrees as high in the tree as possible. This effectively reduces the number of nodes that must be read, examined, and processed.

As nodes are recognized by the scheduler as being visible and at the correct level of detail (LOD), they are placed on the need list. Rather than appending the node at the end of the list, it is actually inserted into the list according to its distance from the viewpoint. This ensures that the nearest, and presumably most important, objects will be processed first by the object processors - a precaution in case the allotted time for all frame processing should expire before the frame is complete.

The algorithm was implemented in Pascal to facilitate manipulation of the linked lists. The implementation consists of under five hundred lines of code, which is very small given the complexity of the algorithm. The small size and modularity enhance the maintainability of the code.

5.8.2 ALTERNATIVES AND FUTURE IMPROVEMENTS

An alternative to the current design would be to restrict the database to a standard tree structure in which all nodes have the same number of children. This would simplify the algorithm somewhat, but would practically remove all flexibility from the actual database. In the current design, additional nodes, such as models, can be added to the tree with little effort. If the tree were standard, addition of nodes might require substantial rearrangement.

Several improvements to the current algorithm are also contemplated. As mentioned in Section 5.6, changes in the field-of-view set (the set containing all visible nodes in the current frame) should be performed in the scheduler. This

would both remove costly work from the Memory Manager and lend more consistency to the overall design.

Since the FOV set changes are small from frame to frame, one could save the root node of the FOV set. Scheduling of the next frame could then begin just above this point in the tree. This would remove the necessity of always traversing the database tree from its root node.

Lastly, the ordering of nodes in the lists by distance is somewhat arbitrary. We plan to experiment and collect statistics on various schemes that may provide better performance. The present scheme was based on forward flight. Other schemes can be easily implemented.

6.0 GRAPHICS ALGORITHM RESEARCH

Part of the effort during this two-year second phase was directed toward investigation of current technology in graphics algorithms. Since the graphics environment is a dynamic one, this commitment to graphics algorithm research was deemed essential to optimize the current CIG system and maximize future efforts. This section presents a summary of the research that was conducted and the implementations that ensued from our study in the fields of fractals, curved surfaces, transparencies, texture mapping, and level-of-detail control.

6.1 FRACTALS

6.1.1 INTRODUCTION

Research of fractals was undertaken to investigate alternate methods of representing the terrain and models necessary for nap-of-the-earth imagery. Fractal representation of an object would allow the generation of high-complexity, realistic images of terrain and natural phenomena at relatively low modeling and storage costs. Fractal methods of representation also would ensure significant data compression over conventional methods, and would allow unlimited LOD control due to the procedural definition inherent in fractals.

6.1.2 DEFINITION

The concept of fractals was introduced by Benoit B. Mandelbrot (Mand77). Mathematically, a fractal is defined as a set for which the Hausdorff-Besicovitch dimension strictly exceeds the topological dimension (Mand71). For an in-depth mathematical discussion of fractals, the following references should be consulted: (Berr80), (Besi37), (Hutc81), (Mand77), (Mand82a), and (Mand82b).

Intuitively, fractals are shapes in which increasing detail is revealed with increasing magnification. The structure revealed at each higher level of magnification is similar to the form displayed at lower levels of magnification. This property is known as self-similarity.

Fractal dimension is a measure of the variability of the fractal object. A fractal curve will have a fractal dimension between one and two. If the dimension is near one, the fractal curve will be relatively smooth and closely resemble a one-dimensional straight line. If its fractal dimension is close to two, the curve will be highly irregular and nearly planar.

Since their introduction, fractals have been used for a variety of applications in CIG. They have been employed to generate objects and natural phenomena such as trees, clouds, turbulence, galaxies, noise, and the solar spectrum. They have also been used to generate realistic surface texture for islands, water, mountains, and polar ice caps (see Figure 66).



Figure 66. Mt. Rainier Enhanced by Fractals

6.1.3 GENERATION OF FRACTAL SURFACES

Fractional Brownian motion (fBm) is a term that denotes a family of one-dimensional Gaussian stochastic processes (Mand82b). This function, extended to two dimensions, is the foundation of present terrain simulation. A number of methods have been published for calculating discrete approximations to fBm. The three basic algorithms are the shear displacement process (Mand75), the modified Markov process (Mand71), and the inverse Fourier transformation (Mand77). Although these methods have solid mathematical foundations, their orders of time complexities are no better than $O(N \log(N))$. Fournier, Fussell, and Carpenter (FFC) (Four82) have presented an algorithm which approximates fBm with a time complexity of $O(N)$. Using this algorithm, Carpenter created the film Vol Libre (Carp80).

The FFC algorithm recursively subdivides a line segment and generates a scalar displacement value at its midpoint. This displacement is statistically proportional to the length of the line segment. With this new midpoint, two new line segments are created which replace the original line segment. The algorithm then continues to recursively subdivide these new segments until some termination criteria such as maximum level of recursion or length of current segment is reached.

When the FFC algorithm is extended to a two-dimensional surface, the edges of a polygon are subdivided and the midpoints are connected to form new polygons (see Figure 67). A disadvantage of this extension is that each interior midpoint is computed twice since the interior polygons share common edges. Therefore, this algorithm exhibits a time complexity of $O(2N)$.

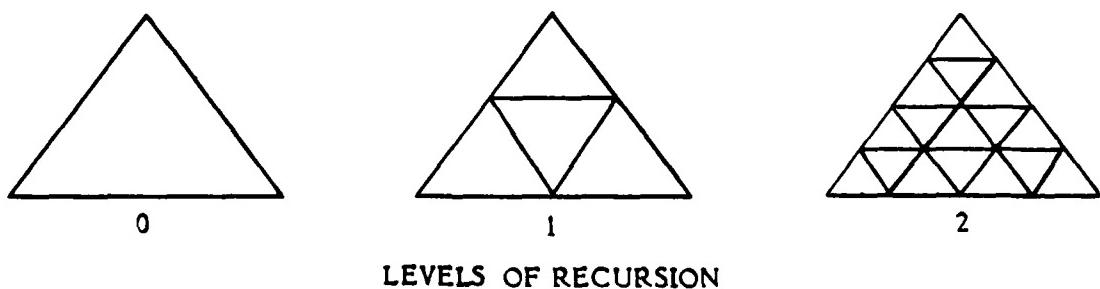


Figure 67. Triangle Subdivision

Fractal research conducted for DARPA has resulted in a new two-dimensional FFC algorithm, written in PASCAL, that exhibits a time complexity of $O(N)$ (see Table 17). In this algorithm, a temporary tree is generated that passes all midpoints created on a shared edge to the adjacent polygon. This allows the computation of interior midpoints to occur only once.

Table 17. Two-Dimensional Fractal Generation Algorithm
(Continued on following page)

The routine "new" allocates dynamic memory for a vertex.
The routine "dispose" deallocates dynamic memory for a subtree.
The routine "terminate" returns "true" when recursive subdivision should be terminated.
The routine "random-gauss" returns a random Gaussian variable with mean 0.0 and variance 1.0.

```
program fractal_tri(input,output);

const
    h    = ???; (* The value of "h" determines the fractal dimension. *)
    scale = ???; (* User defined scale factor. *)

type
    vtx_ptr = ^vtx;(* pointer to a vertex *)
    vtx = record    (* vertex *)
        x,y,z: real;
        left,right: vtx_ptr;
    end;
var
    v1,v2,v3: vtx;
    t1,t2,t3: vtx_ptr;
    std,ratio: real;

(*****)
(* Subdivide fractal triangle. *)
procedure fractri(var v1,v2,v3: vtx; var t1,t2,t3: vtx_ptr; std: real);
    var tm1,tm2,tm3: vtx_ptr;
    (*-----*)
```

Table 17. Two-Dimensional Fractal Generation Algorithm
 (Continued on following page)

```

procedure generate_midpoint(var v1,v2: vtx; var m: vtx_ptr);
var z_mean: real;
begin (* generate_midpoint *)
    if m = nil
    then begin (* create a new midpoint vertex *)
        new(m);
        with m^ do
            begin (* generate the midpoint *)
                x := (v1.x + v2.x) * 0.5;
                y := (v1.y + v2.y) * 0.5;
                z_mean := (v1.z + v2.z) * 0.5;
                z := z_mean + std * random_gauss(x,y);
            end; (* generate the midpoint *)
    end; (* generate_midpoint *)
begin (* fractri *)
    tm1 := nil; tm2 := nil; tm3 := nil;
    if not terminated(v1,v2,v3)
    then begin (* generate new midpoints & call recursively *)
        generate_midpoint(v1,v2,t1);
        generate_midpoint(v2,v3,t2);
        generate_midpoint(v3,v1,t3);

        std := std * ratio;

        fractri(t1^.t2^.t3^.tm1, tm2, tm3, std);
        fractri(t1^.t3^.v1, tm3, t3^.left, t1^.right, std);
        dispose(tm3);
        fractri(t2^.t1^.v2, tm1, t1^.left, t2^.right, std);
        dispose(tm1);
        fractri(t3^.t2^.v3, tm2, t2^.left, t3^.right, std);
        dispose(tm2);
    end;
end;

```

Table 17. Two-Dimensional Fractal Generation Algorithm
(Concluded)

```
(* Exchange the left & right subtrees. *)
exchange(t1^.left, t1^.right);
exchange(t2^.left, t2^.right);
exchange(t3^.left, t3^.right);

end; (* call recursively *)
end; (* fractri *)

begin (* main program *)
ratio := 2 ** (-h);
std := scale * ratio;

t1 := nil; t2 := nil; t3 := nil;
fractri(v1,v2,v3,t1,t2,t3,std);

end. (* main program *)
```

6.1.4 FUTURE DEVELOPMENTS

Although present development favors texture mapping over fractals as a means of enhancing scene content, fractals can create spectacularly realistic scenes. However, substantially more research is needed before this technology can be considered mature. Future research will address such issues as methods of abstracting fractal parameters from real data, methods to match fractal and real representations, and innovative methods to render fractals.

6.2 CURVED SURFACES

Curved surface research was undertaken primarily to gain familiarity with non-polyhedral surface representation and rendering techniques. The first objective for the project was to identify an optimal mathematical surface representation for general application. The second objective was to implement a graphics processor that would render this surface form into a solid-shaded computer image.

A surface representation was considered to be optimal if it could be easily and intuitively manipulated, if it was flexible enough to represent a wide variety of shapes, if it ensured significant data compression over polyhedral representation, and if its rendering algorithm was computationally feasible and efficient. The surface forms which were investigated fell into four classes - quadrics, superquadrics, B-splines, and Beta-splines.

6.2.1 QUADRICS

Quadrics is the class of surfaces defined by quadric equations. A rudimentary graphics processor that rendered quadric surfaces was implemented prior to current curved surface research. This processor accepted input in the form of coefficients of a quadric equation. A ray-tracing algorithm was used to extend a vector from the viewpoint, through each screen pixel, to the quadric surface. The quadric equation was evaluated to determine the distance or depth from the viewpoint to the quadric surface along this vector. This information was then used to build a z-buffer image for display (see Plate VII).

Quadric equations allow surfaces to be represented exactly without approximation. Furthermore, a quadric equation can be easily evaluated, making it an ideal candidate for rendering by ray tracing. Evaluation by ray tracing opens the door to realistic shading, allowing lighting effects such as refraction and specular reflection to be easily and accurately simulated. However, quadric surfaces are very limited in scope. They can only represent a small subset of the shapes necessary for most applications. Also, these surfaces are inflexible and difficult to manipulate interactively.

6.2.2 SUPERQUADRICS

These limitations of quadrics have helped to spawn a new class of quadric surface representation called superquadrics (Barr81). Superquadrics are mathematical solids based on a spherical product of two of the four parametric forms of quadric surfaces - ellipsoids, hyperboloids of one piece, hyperboloids of two pieces, and toroids. The superquadric parametric equations are composed of trigonometric functions raised to an exponent. The exponents are used as squareness parameters to give more flexibility to the resulting shape. Formulas called inside-outside functions exist for each type of superquadric and enable one to determine where any given point lies in relation to the superquadric surface. Because of these functions, superquadrics can be manipulated by means of solid Boolean operations, such as union, intersection, and subtraction. Angle-preserving transformations can be applied to bend and twist a superquadric without compromising volume, surface area, or arclength.

Superquadrics seem to satisfy the criteria for manipulation, flexibility, and data compression. However, this surface form is relatively new and most of the information available is theoretical. No studies have been published on its implementation as a surface representation or the techniques to be utilized in rendering this form. More concrete information on implementation strategies is needed before judgment can be passed on this surface form.

6.2.3 B-SPLINES

The third class of surface forms - B-splines (Gord74, Debo78) - are piecewise parametric polynomials that combine control points, parameter space values called knots, and a set of blending functions to define a surface (Figure 68). Because of the mathematical construction of a B-spline surface, the control points have the ability to approximate the surface and emulate its behavior. For most manipulations, the control points can replace the spline surface and the cumbersome equation that defines it without loss of accuracy.

The control points of a B-spline surface carry shape information about the surface and are predictably tied to the surface by the blending functions. When the position of a control point is changed, the surface affected by that control point is

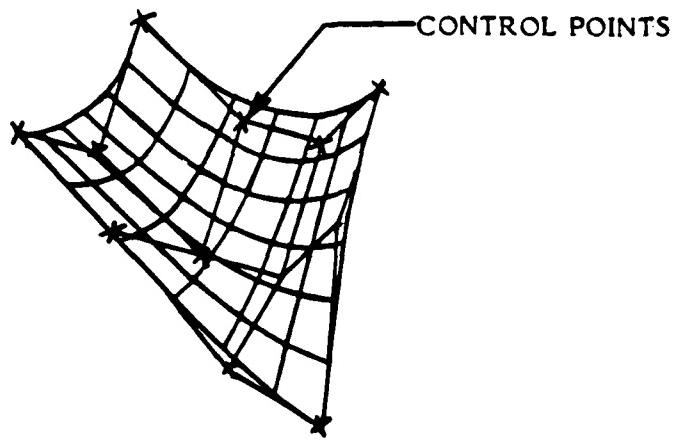
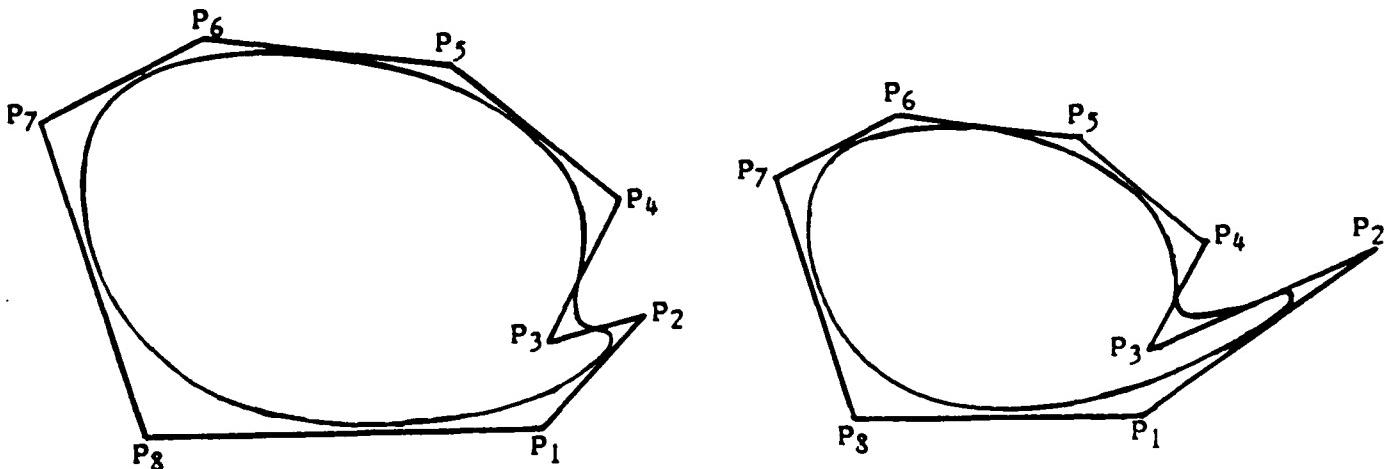


Figure 68. B-Spline Surface and Control Point Grid

revised in an intuitive manner relative to the movement of that control point (Figure 69). This relationship makes B-spline surfaces a good choice for any application where interactive manipulation is a concern.



NOTE EFFECT ON CURVE OF MOVING P₂

Figure 69. Two Closed B-Spline Curves

Two characteristics of B-spline surfaces expand their flexibility, and hence their ability to represent a wide variety of shapes. The control points of a spline surface usually exert local control over the surface, meaning that each control point only affects a small local subset of the entire surface. This allows a small section to be intricately manipulated without affecting the shape of any other part of the surface. Also, control points and knots can be added at will to a B-spline surface representation without affecting the degree of the spline equation. This allows control points and knots to be concentrated in areas where high variability is desired.

Because the control points can actually substitute for the spline polynomial as an approximate representation, a spline surface can be stored as a set of control points and its accompanying knot values. Similarly, a rendering algorithm for B-spline surfaces need only concern itself with the control point grid and knot values. The underlying spline equation never needs to be evaluated or even identified.

In addition, B-spline surfaces exhibit two properties that are useful for any surface form. First, the convex hull property dictates that the surface always lies within the convex polyhedron formed by joining the control points. Thus, one always has a bounded surface with identifiable minimums and maximums. Second, the variation-diminishing property guarantees that the control point approximation will be at least as smooth as the primitive function. This ensures the stability of the approximation and its refinements.

6.2.4 BETA-SPLINES

The last surface form considered was Beta-splines (Bars83). There are two classes of Beta-splines. Uniformly-shaped Beta-splines are a generalization of uniform cubic B-splines. The parametric first- and second-degree continuity of uniform B-splines is replaced by geometric continuity over the curve or surface, expressed by a continuous unit tangent and curvature vector. This allows the specification of two extra global parameters - tension and bias. These two parameters allow the curve or surface to be finely tuned globally without loss of continuity at joints. The second class is the continuously-shaped Beta-splines. These splines, which are in fact a restricted form of a quintic Hermite interpolation, take uniformly-shaped

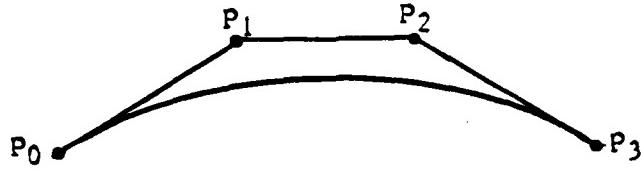
Beta-splines a step further. They allow local specification of tension and bias at joints, without destroying geometric continuity. Both Beta-splines retain the convex hull and variation-diminishing properties of B-splines and allow local control of shape through the control points.

Beta-splines seem to retain all the advantages of B-splines with the addition of the increased flexibility allowed by the tension and bias parameters. However, Beta-splines are also a new technology, and, as a result, are lacking in implementation studies and strategies. Before this surface form can be endorsed, more practical information is needed.

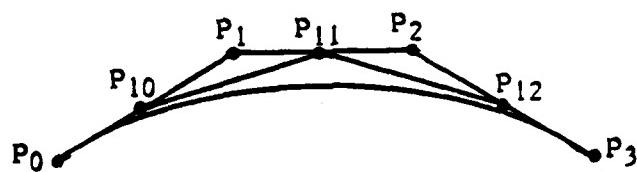
6.2.5 IMPLEMENTATION

Based on the preceding information, B-splines emerged as the current front runner in the search for an optimal surface representation. The form that was actually implemented was a special case of B-spline surfaces called Bezier patches. Through knot and control point insertion, any B-spline surface can be specialized into a series of adjacent Bezier patches. Bezier patches retain most of the advantages of B-spline surfaces, including the convex hull and variation-diminishing properties, and the intuitive relationship between the spline surface and its control points. However, Bezier patches use the Bernstein blending functions and restrict the parameter space between 0 and 1. This combination reduces the spline equation to a purely polynomial form that allows a simplification of methods when manipulating the control points. However, a loss of flexibility accompanies this gain in simplicity. The advantage of local control is lost, meaning that a revision of any control point causes a global change in the surface.

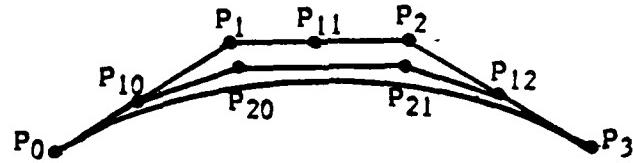
The design and implementation of the graphics processor evolved naturally from the surface form chosen as its basis. The choice of Bezier patches dictated the choice of subdivision as the rendering algorithm for this processor. Subdivision (Cohe80, Debo78) is a fast, efficient rendering algorithm that utilizes the shape information contained in the control points of a B-spline surface. The control points are refined by this algorithm into a bilinear approximation of the surface. If the refinement is allowed to continue indefinitely, the surface defined by the control point grid will converge to the underlying spline surface (see Figure 70). Subdivision enables one to achieve a fine approximation through the control points without ever evaluating the spline function.



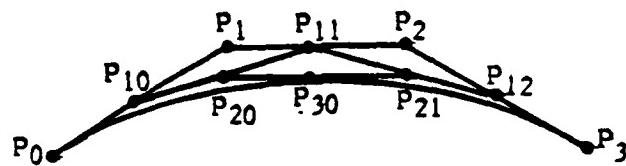
ORIGINAL BEZIER CURVE WITH CONTROL POINTS P_0-P_3



FIRST SUBDIVISION - CONTROL POINTS $P_0, P_{10}, P_{11}, P_{12}, P_3$



SECOND SUBDIVISION - CONTROL POINTS $P_0, P_{10}, P_{20}, P_{21}, P_{12}, P_3$



THIRD SUBDIVISION - RESULTING IN TWO NEW BEZIER CURVES

CONTROL POINTS OF THE FIRST CURVE - $P_0, P_{10}, P_{20}, P_{30}$

CONTROL POINTS OF THE SECOND CURVE - $P_{30}, P_{21}, P_{12}, P_3$

Figure 70. Subdivision on Bezier Curves

The subdivision algorithm that was implemented was a version of the Lane-Riesenfeld algorithm (Lane80) which subdivides a patch in two directions simultaneously, creating four new patches. It was felt that this blind bivariate subdivision would result in unnecessary computation when a patch was essentially flat in one direction, for example, a cylindrical surface. Therefore, an intelligent subdivision algorithm was implemented that subdivided only in the direction where more refinement was necessary. If both directions needed subdivision, the algorithm was executed twice in different directions.

The actual mechanics of the subdivision algorithm consist of utilizing the original set of control points that define a Bezier patch over a parameter space to generate two new sets of control points. Each set defines half of the original Bezier patch over respective halves of the original parameter space. The method used to generate the new points is similar to a forward-differencing algorithm. Each new control point is a weighted sum of some subset of the original control points. The two new sets of points are a closer approximation to the surface and become eligible for further subdivision. This process can go on indefinitely and will eventually lead to convergence (see Figure 71).

Since indefinite subdivision is not feasible, stopping criteria must be defined. For this reason, subdivision algorithms are paired with one or more tests for flatness to determine when a good piecewise bilinear approximation has been achieved. Such tests are specifically used to test for flatness and planarity of the control point grid.

The first test implemented was developed by Jeremy Jaech of the Boeing TIGER project (Jaec82). Each control point grid was viewed as a series of parallel curves. For each control point curve, a line segment was formed by joining the curve endpoints. The Euclidean distance from the interior control points to the line segment was measured (see Figure 72). If that distance fell within a predetermined tolerance for all curves in the control point grid, that grid was determined to be essentially flat.

The second test was developed by Larry Matthies of the University of Waterloo in his master's thesis (Matt80). It makes use of the fact that the four cornerpoints of a Bezier control point grid are guaranteed to lie on the underlying spline surface. In this test, the four cornerpoints are compared to determine if all fall within a

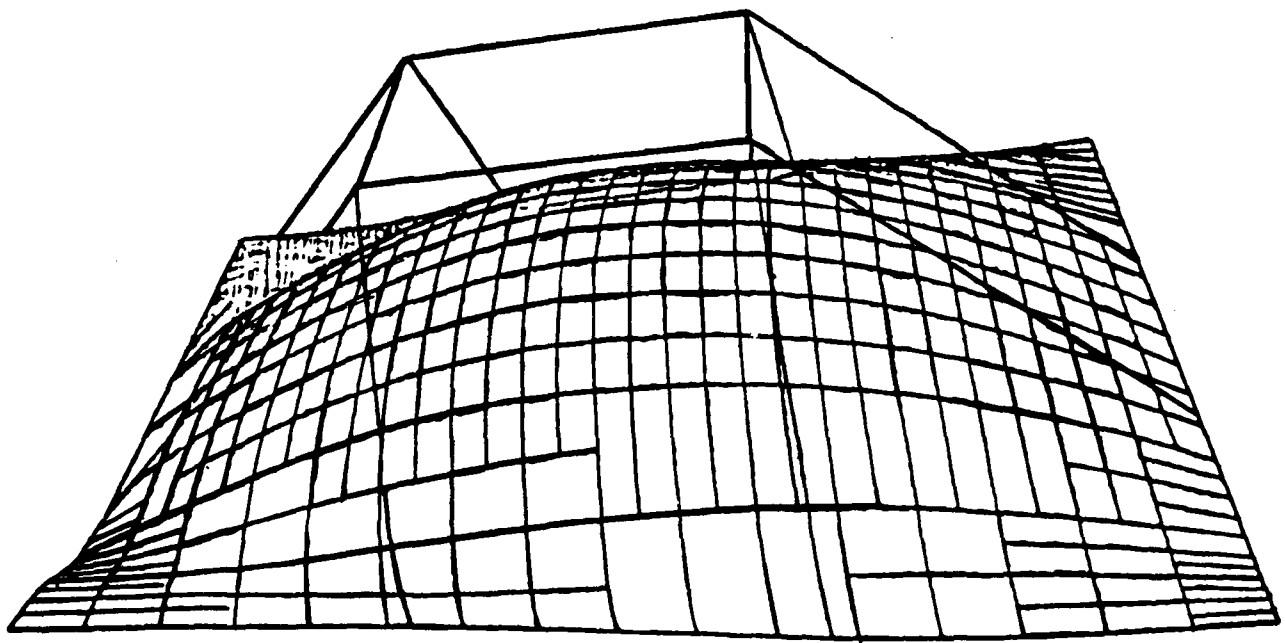
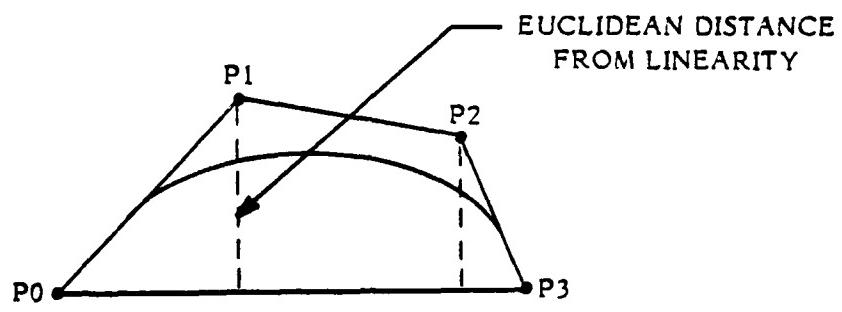
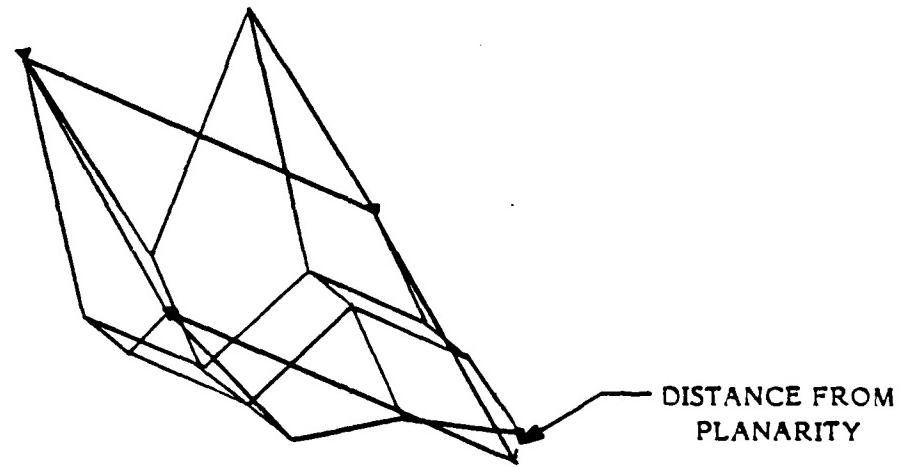


Figure 71. A Bicubic Bezier Patch as Rendered by the Subdivision Algorithm and its Original Control Grid



Flatness test - Testing for linearity of
a control point curve



Planarity test - Testing for planarity of the
four corner control points

Figure 72. Tests for Flatness

predetermined tolerance of being planar (see Figure 72). If they pass this test, it signifies that the grid cornerpoints essentially define a plane on the spline surface. If a grid simultaneously passes the preceding flatness test, it ensures that the grid is a good piecewise bilinear approximation to a planar section of the spline surface.

Once a control point grid has been refined into a good approximation, it must be converted into pixel information. For this section of the process, it was decided to take advantage of the existing DARPA software and hardware. Since triangles are the medium of the DARPA system, our control point grid was triangulated by constructing diagonals within each quadrilateral of control points. These triangles were then sent through the DARPA system starting with the DARPA pretiler (see Section 5.3) for tiling and display (see Figure 73).

To obtain color information, curve shading was used over the patch. Vertex normals were calculated for each control point in the grid by taking an average of the face normals of all the triangles that shared that control point. These vertex normals were then crossed with the sun vector and combined with color information provided by the user to produce RGB values for the DARPA tilers.

Curve shading over patches did produce minor color discontinuities at patch boundaries. Boundary control points might be shared by as many as four separate patches. Therefore, all the necessary face normals are not available during color calculation, since patches are processed one at a time from a stack. This problem could not be addressed completely due to time and budget constraints. For a future implementation, a pipeline could be constructed where patch boundary face normals are retained to be used in the vertex normal calculations of adjacent patches.

6.2.6 FUTURE DEVELOPMENT

This research was undertaken with the express goal of gaining familiarity with curved surface representation and rendering. The next step would be to redesign our present system with a more flexible, efficient surface form. The choice of an optimal surface form would depend necessarily on the application, with Beta-splines being a suitable choice for an interactive CAD application and superquadrics being the form of choice for static model applications. In either case, the surface form would dictate the implementation of the rendering processor.



Figure 73. Teapot Modeled with Twenty-eight Bicubic Bezier Patches

6.3 TRANSPARENCY

Transparencies are essential for modeling a variety of objects and phenomena. They can be used to simulate atmospheric conditions such as clouds, rain, snow, haze, and fog as well as transparent and translucent surfaces. Transparency algorithms are also useful for blending two levels of detail of an opaque model.

Two methods of producing transparencies have been researched and implemented here - particle systems and the thickness buffer.

6.3.1 PARTICLE SYSTEMS

Particle systems were introduced by William Reeves of Lucasfilm Ltd. (Barn83). Particles are discrete sections of some substance that carry with them velocity and direction information, as well as a lifetime and color. Its application here was primarily to model clouds. The color of a pixel containing a transparent cloud particle is computed according to the following rule (due to Rayleigh and Blinn (Barr81)):

$$\text{New Color} = T \text{ (Old Color)} + (1 - T) \text{ (Cloud Color)}$$

where T is the transparency factor.

This method requires transparent objects to be processed last, since all background color (old color) must be present in the image buffer before new color can be calculated.

T , the transparency factor, is computed by a variety of methods. In our particle system implementation, T is basically an exponential function of the distance of the particle from the viewpoint. It is computed as described below.

A small particle projects to a very small area on the screen. However, the smallest resolution element of the screen is one pixel. Thus, a super-particle is created whose shape adjusts itself to fulfill the following requirements:

- A. It always subtends one pixel on the screen.
- B. Its density is constant.
- C. Its mass is constant.

In this fashion, the properties of a real particle (e.g., a water droplet in a cloud) are distributed over one pixel on the screen.

One can now compute the amount of light transmitted through this super-particle according to Rayleigh scattering:

$$T = e^{-\rho d}$$

where ρ is the density of the particle; and
 d is its thickness.

The density ρ is constant. However, the thickness d varies. The volume of the particle is

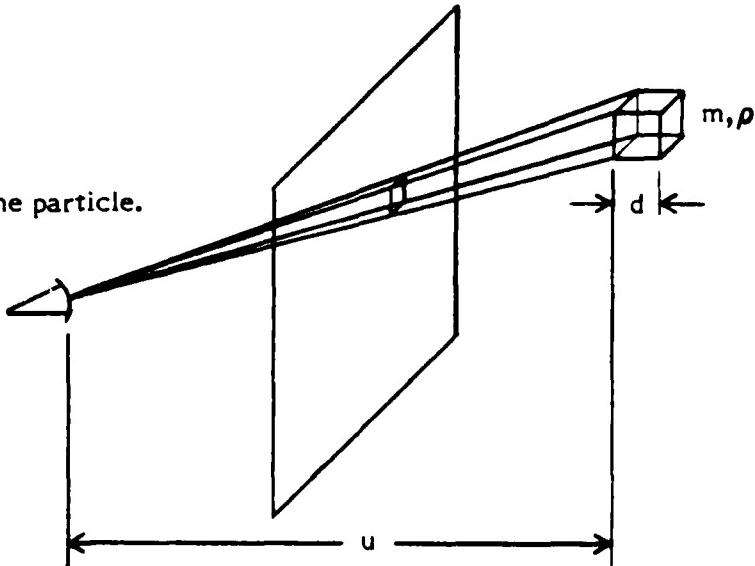
$$V = \frac{m}{\rho}$$

where m is the particle's mass.

The volume can also be calculated geometrically. The width of the particle is 1 pixel. Using the viewpoint-to-screen-space projection equation (see Section 4.2), one finds:

$$V = u^2 d$$

where u is the distance to the particle.



Combining the two equations for V and solving for d, one finds

$$d = \frac{m}{\rho u^2}$$

The transparency factor T is therefore:

$$T = e^{-\frac{m}{u^2}}$$

One can now experiment with various values for m.

The apparent density of the cloud can be increased by spreading each particle over more than one pixel. In the current implementation, the transparency factor is computed for the pixel to which the particle projects, then the same factor is used with appropriate weighting for the eight surrounding pixels. The weighting function is as follows:

$$\begin{matrix} .25 & 0.5 & .25 \\ 0.5 & 1.0 & 0.5 \\ .25 & 0.5 & .25 \end{matrix}$$

The use of this weighting function simulates a macro-particle with a two-pixel diameter.

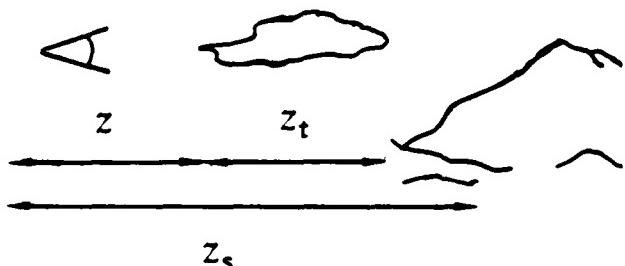
6.3.2 THICKNESS BUFFER

The thickness buffer approach was developed at Boeing. It involves maintaining a second buffer the same size as the depth buffer (see Section 2.0) in which information about the thickness of transparent objects at each pixel is stored. When a triangle is about to be tiled (see Section 4.2), two of four conditions exist:

1. The triangle is transparent or it is opaque.
2. The triangle is a back face or a front face.

If the triangle is opaque, it is tiled normally. Otherwise, the following algorithm is executed:

```
Z := min (Z, Zs(i,j))  
if front face    Zt(i,j) := Zt(i,j) - Z  
else            Zt(i,j) := Zt(i,j) + Z
```



where $Z_s(i,j)$ is the value in the depth buffer for pixel (i,j)
 $Z_t(i,j)$ is the value in the thickness buffer for pixel (i,j)
 Z is the depth to the transparent object or the nearest solid object, whichever is nearer.

This algorithm is based upon the fact that the thickness of the transparent object is equal to the distance from the viewer to the back of the transparent object minus the distance to its front. In addition, the distances are clipped if a solid object is nearer than the transparent object. Note that this algorithm behaves correctly under the following adverse circumstances:

- A. The opaque object intersects the transparent object.
- B. The opaque object is in front of the transparent object.
- C. The transparent object envelopes the viewpoint.
- D. There are multiple transparent objects.
- E. There are concave transparent objects.
- F. There are embedded transparent objects.
- G. The surface of the transparent object intersects itself.

After all triangles have been processed, the image buffer is altered according to the contents of the thickness and depth buffers. The variable T in the above equation is now related just to the thickness of the intervening transparent object(s).

When clouds are modeled as objects with faces, as in the thickness buffer approach, surface reflection can be added to produce surface highlights.

6.3.3 PROBLEMS

There are difficulties associated with each method. While particle systems can accurately model the physical processes that occur in clouds, they are computationally very expensive. The number of particles required for realistic density in a cloud image is on the order of ten thousand. This could be reduced by increasing the size of the particles substantially. The effective particle sizes currently in use are measured in inches. An increase of the effective size to meters could further compress particle numbers, but could have a detrimental effect on the realism of the results.

The thickness buffer approach requires a large amount of memory. The thickness buffer is the same size as the depth buffer, 512x512x4 bytes. It also requires two passes through the data, one to compute the thicknesses and another to alter the image buffer. This method has been discarded because of its inefficiency and disproportionate use of memory.

6.3.4 FURTHER DEVELOPMENTS

Transparencies will become an important feature of any future CIG flight simulation system. Research is ongoing to further develop the transparency concept. In particular, recent work has indicated that transparency algorithms can be developed to eliminate excessive memory requirements and to allow arbitrary processing of transparencies with opaque objects.

6.4 TEXTURE AND COLOR MAPPING

6.4.1 INTRODUCTION

A significant problem encountered in generating realistic computer images is the lack of textural detail. Frequently, a scene is so smoothly shaded that there is little variation in color. Images with this subtle shading lack realistic textural detail. Since texture may have a very irregular structure, it is very difficult to generate a natural-looking texture using mathematical functions.

Attempts to modify the geometry of the terrain or model data have been made using various texturing techniques. Geometry parameters, such as position and surface normals, may be randomly or functionally perturbed to produce texture (Blin78a). With this process, the texture effect is achieved by the shading of the modulated geometry. However, such functions are computationally intense, and do not always produce the desired textural appearance. Often, the result will be an unrealistic or recurring texture pattern.

An alternative to altering the geometry data is to project a texture map, which may be an actual photograph of some textural information, onto the terrain or model (Fole82). Brick walls, grass, and trees are examples of the types of textures that may be mapped to achieve truly realistic results. There are, however, problems associated with this texture mapping technique. The difficulties include storage requirements, computational requirements, and the handling of level of detail. The techniques used to solve these problems will be discussed in this section.

6.4.2 DESIGN CRITERIA AND OBJECTIVES

The main objective in texturing is to achieve realism. To accurately simulate texturing, generated textures must be as detailed as those occurring naturally.

Another objective is to keep the texture map independent of the geometric database. This allows a single texture to be applied to any number of models. It also disassociates the resolution of the texture map from the resolution of the

underlying geometry. Thus, much visual detail can be added to a simple geometric model by using a highly detailed texture map.

The processing speed of the texture algorithm is always of concern. One should therefore avoid performing costly computations for every pixel.

Finally, the texture should be applied in a fashion which avoids aliasing effects. Specifically, scintillations, Moire patterns, and blurring are to be avoided.

6.4.3 DESCRIPTION OF THE TEXTURE MAPPING TECHNIQUE

The texture mapping technique presented here is basically an orthographic projection of a texture map to the geometric model (Figure 74). It involves three parts:

- A. The determination of the color map index in an efficient, iterative manner.
- B. The use of a mip map memory arrangement to efficiently store downsampled texture maps and handle the levels of detail of these maps.
- C. The calculation of the level-of-detail variable, and the blending of the colors in the maps.

First, an explanation of the texture map indexing scheme will be given. This will be followed by a discussion of the mip map memory arrangement and how the indices are used to find the proper values in the mip map. And finally, the calculation of the LOD variable and how it is used will be described.

6.4.3.1 TEXTURE MAP INDEX EQUATIONS DERIVATIONS

When tiling, only the screen coordinates, (i,j) , and the depth, $d=1/U$, are known. By combining the projection and transformation equations of the screen space with those of the map space, a direct transformation from screen space to map space can be derived. This transformation is then reformulated into a form suitable for an iterative algorithm. This section fully develops these concepts.

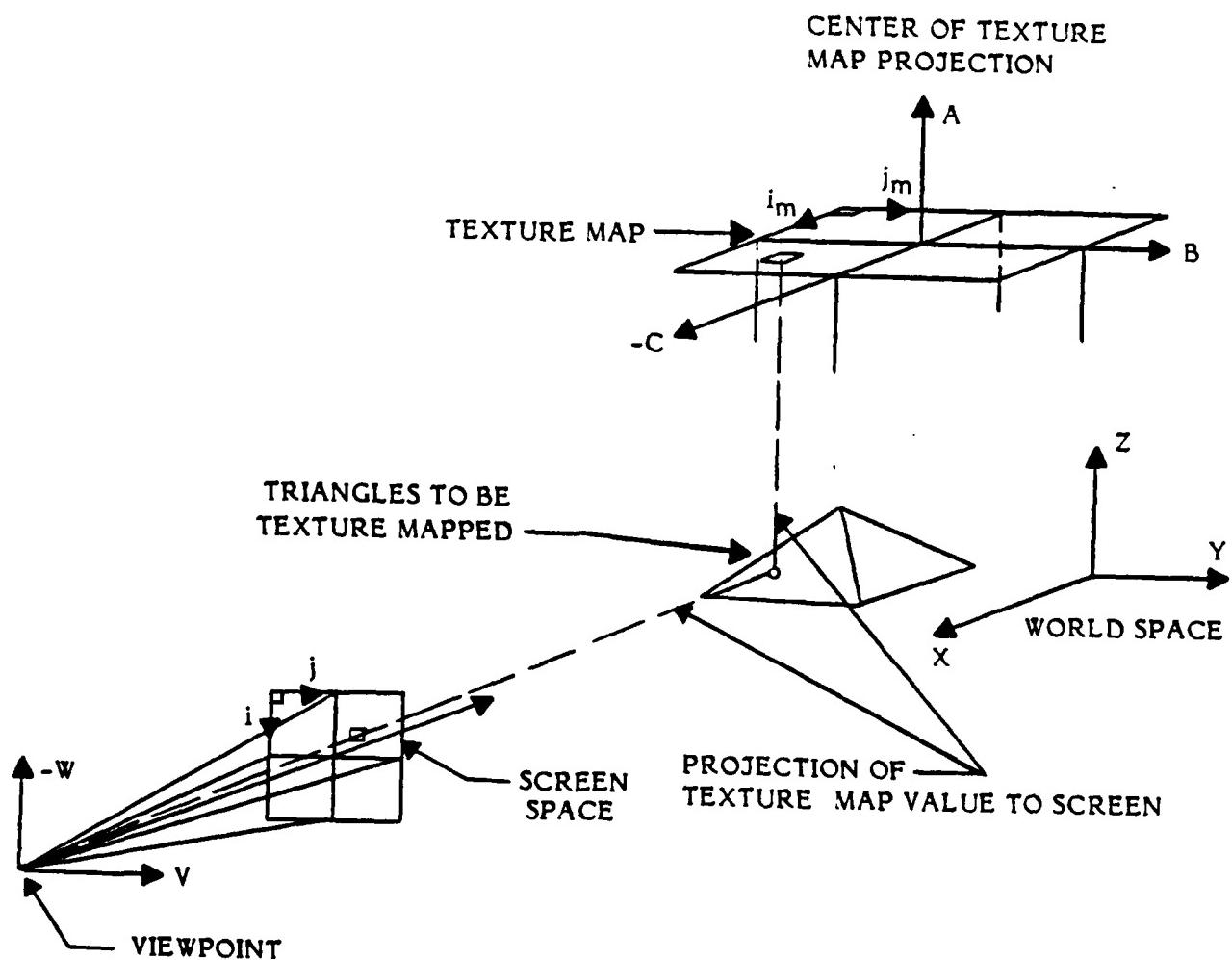


Figure 74. Projection of Texture Map

The texture map is projected orthographically (see Figure 74). The 3-space to 2-space projection equations can be written as:

$$i_m = c + i_{mh} \quad j_m = b + j_{mh}$$

where (i_m, j_m) are the map indices;
 (a, b, c) is the 3-space as defined in Figure 74;
 (i_{mh}, j_{mh}) are the coordinates of the map's origin.

The transformation from (a, b, c) space to (U, V, W) (viewpoint) space is accomplished via (X, Y, Z) (world) space:

$$\begin{aligned} (U, V, W) &= (X, Y, Z)S & (a, b, c) &= (X, Y, Z)R \\ (a, b, c) &= (U, V, W)S^{-1}R = (U, V, W)T \\ a &= UT_{11} + VT_{21} + WT_{31} + T_{41} \\ b &= UT_{12} + VT_{22} + WT_{32} + T_{42} \\ c &= UT_{13} + VT_{23} + WT_{33} + T_{43} \end{aligned}$$

The viewpoint space to screen projection equations are now used, as well as the expression for depth (see Section 4.2.2.3):

$$V = U(j - j_h) \quad W = U(i - i_h)$$

$$U^{-1} = d = d_o + \frac{\partial d}{\partial i} (i - i_o) + \frac{\partial d}{\partial j} (j - j_o)$$

After substitution and considerable simplification, the equations of the map coordinates take the form:

$$i_m = \frac{i_{mo} + i_{mi}(i - i_o) + i_{mj}(j - j_o)}{d} + i_{mh}$$

$$j_m = \frac{j_{mo} + j_{mi}(i - i_o) + j_{mj}(j - j_o)}{d} + j_{mh}$$

where i_{mo}, i_{mi}, i_{mj} and j_{mo}, j_{mi}, j_{mj} are constants.

These equations can be used for an iterative tiling algorithm in the same fashion as the depth, color, and dot products are used in the tiler (see Section 4.2.2.4). The texture mapper performs the following calculations in addition to those performed by a tiler:

- A. During the set-up calculations for the triangle, it must compute i_{mo} , i_{mi} , i_{mj} , j_{mo} , j_{mi} , and j_{mj} .
- B. For each pixel the partial sums of i_{mi} and j_{mi} (motion in i-direction) or of i_{mj} and j_{mj} (motion in j-direction) must be updated.
- C. The total sums must be divided by depth and added to the map center.

This approach to calculating map space coordinates clearly parallels the approach taken in the tilers. It thereby prepares the way for a hardware implementation of the texture mapper which uses basically the same architecture as a tiler. The division, which is in general slow and costly, is made feasible by the limited precision required.

6.4.3.2 MIP MAP MEMORY ARRANGEMENT

To prevent aliasing effects for distant texture maps, a method of processing a low-resolution version of the map must be devised. To minimize the on-line computational costs, it was decided to use prefiltered versions of the texture map with the original map defined as a 128x128 array. One must now find an easy way to store the texture map with its prefiltered descendants.

A compact way to store the original and downsampled texture maps is with mip maps. This method, developed in parallel at Boeing and at the New York Institute of Technology, is described by Lance Williams in (Will83). "Mip" is an acronym from the Latin phrase "Multum in parvo," meaning "Many things in a small place". The format of this memory arrangement is shown in Figure 75. The unsampled, highest resolution components of the map take up 75% of the memory space, while all the downsampled versions take up only 25% of the allocation. The actual size of the mip map is a 256x256x1 byte array. Each mip map represents seven levels of detail.

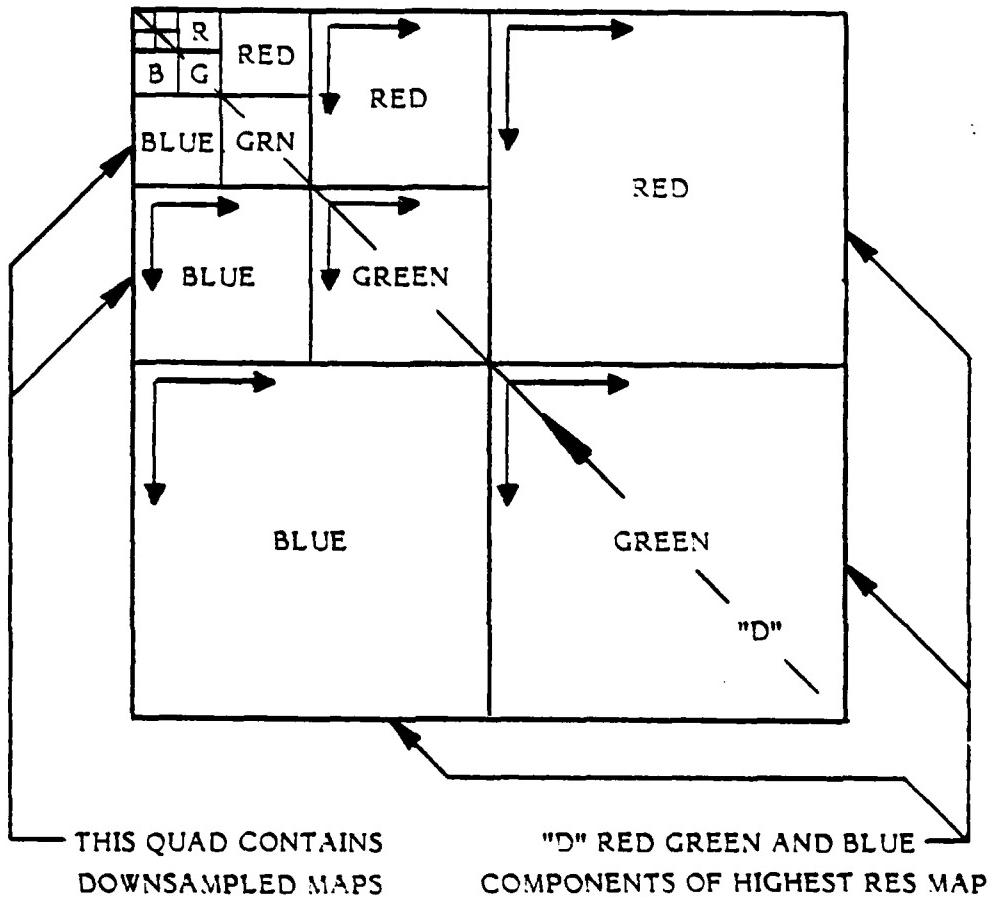


Figure 75. Memory Arrangement of Color Mip Map

'D' is the variable used to index and interpolate between levels of maps. All downsampling of maps done off-line.

Each triplet of downsampled maps is associated with a value of the LOD variable D. This LOD variable is used to decide which level-of-detail maps will be used.

Changing the level of detail in a mip map is very simple. One need only scale the map indices by a power of two; in other words, the level of detail can be changed by right or left-shifting the map indices.

6.4.3.3 LOD VARIABLE CALCULATION AND BLENDING TECHNIQUE

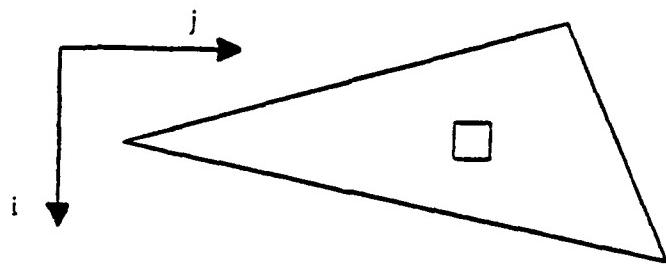
The LOD variable representation is shown in Figure 76. The top figure shows a pixel being tiled in a triangle. The lower figure shows the "pseudo projection" of a pixel in the highest resolution mip map. The dimensions of the pixel projection are determined from the step size in the highest resolution map for one step in screen coordinates ($D = 3.5$ in the example in Figure 76). The dimension of the pixel projection is used as the LOD variable previously mentioned, and used to choose the two mip map levels whose color values will be used (see Figure 77).

The calculations to extract the final color value from a mip map are illustrated by the example in Figure 77:

- A. The indices are calculated in the highest resolution mip map ($D=1$).
- B. The LOD value D is calculated from the "pseudo pixel projection".
- C. The D value ($D = 3.5$) determines that map levels 2 and 4 are used.
- D. The map indices are shifted the appropriate number of bits to access the proper levels of the map.
- E. The four neighboring color values in the two map levels nearest the D value are averaged with bi-linear interpolations.
- F. The two intermediate colors are averaged with a linear interpolation based on D to get the final color. (The last two steps are performed once for each color component.)

The interpolations are used to prevent aliasing and LOD transition problems. The interpolation among the four nearest neighbors in each mip map level prevents scintillations; it also prevents the map cell boundaries from becoming apparent when the map is very near the viewpoint. The interpolation between mip map

TILING A TRIANGLE



APPROXIMATE PIXEL PROJECTION INTO MIP

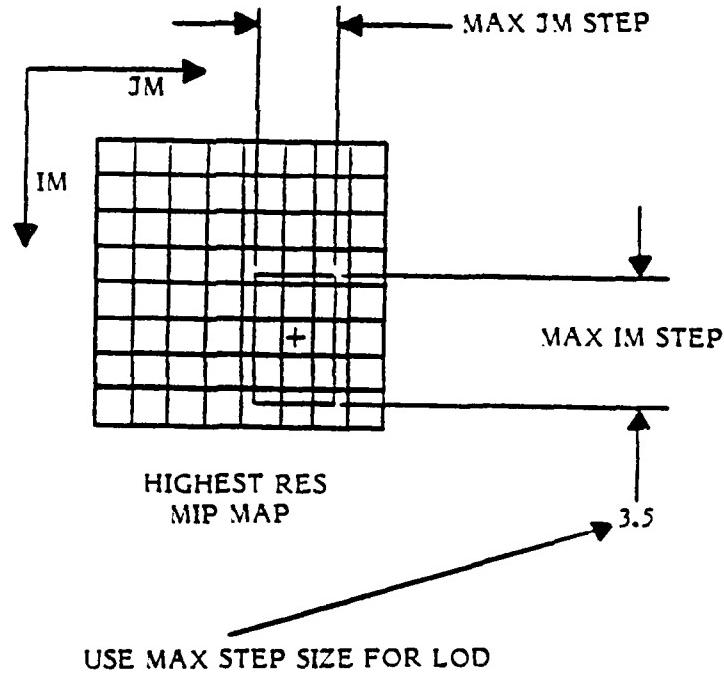


Figure 76. Calculating Mip LOD

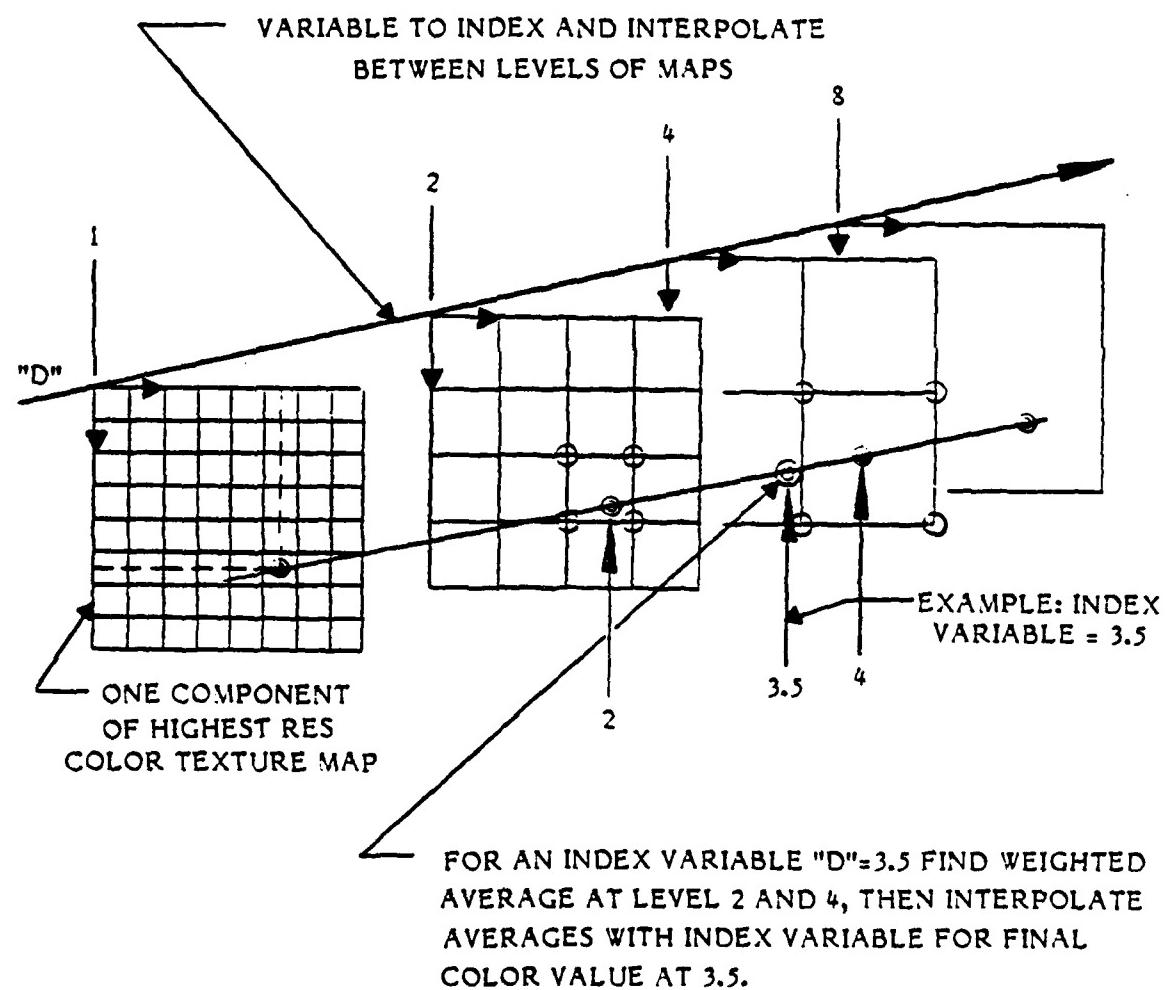


Figure 77. Extracting Final Color Value from a Mip Map

levels prevents LOD transitions from becoming apparent. Experimentation verified that the interpolations are effective as well as necessary.

Note that the LOD is computed on a pixel by pixel basis. This allows a map which stretches from very near to very far (e.g., a texture-map runway) to be shown at the correct LOD throughout the map.

6.4.4 ALTERNATIVE TEXTURE METHODS

Two alternative methods for generating surface texture were considered; both were discarded because they did not satisfy the requirements as well as the texture map technique did. These alternatives, and reasons for not using them, are discussed below.

6.4.4.1 PERTURBATION OF SURFACE NORMALS

One method of surface, texturing is to perturb surface normals of the geometry before it is used in the shading model. Although the visual effect is that of a rough surface, the actual geometric data is not changed. An algorithm must be developed to select and perturb surface normals; alternatively, a look-up table detailing a pattern of perturbations can be used (Blin78a). To change the texture, the algorithm must be modified or a new texture pattern look-up table must be generated. Use of these texture schemes only affects the shading, so coloring must be applied separately.

It was concluded that this texturing method was not flexible enough. The necessity of separating coloring and texturing makes this method unattractive. In contrast, using the texture map technique, maps of any texture pattern may be mapped onto any surface. The map incorporates both coloring and shading information. This provides the flexibility necessary for all types of texturing, from mapping aerial photography to texturing and coloring models.

6.4.4.2 FUNCTIONAL MODULATION OF INTENSITY

A technique used by the Grumman Aerospace Corporation to generate surface texture is functional modulation of intensity (Stei83). A mathematical function is

used to determine the amount of modulation in intensity. The parameters of the function are varied to simulate the specific texture of the model or terrain being represented. A fairly complex function must be used to achieve enough flexibility to model differing types of texture. The texturing function must then be evaluated at every pixel; this is very computationally intense. This method also does not incorporate color variations, and therefore tends to yield scenes with little visual content. This method was rejected because of its poor performance and lack of flexibility.

6.4.5 DIFFICULTIES WITH THE TEXTURE MAPPING TECHNIQUE

Two problems with the texture map technique have surfaced: processing speed and memory usage.

The problem with processing speed is caused primarily by the blending scheme. Interpolating eight color values per pixel is time-consuming. This speed problem will be resolved in a hardware implementation, since the blending of colors from the different LOD texture maps can be done in parallel. Simplification of the blending procedure in hardware may also improve processing speed.

Since the texture mapping technique is highly memory-intensive, memory limitations can become a problem. This is true especially if many high resolution texture maps are used for the terrain and models. This problem can be avoided if efficiency is used in building the texture maps, using low resolution maps when possible, and limiting resolution if necessary.

6.4.6 FUTURE ENHANCEMENTS AND DEVELOPMENT

Given the ability to process variable resolution and variable scale texture maps, an extensive and flexible texture map database generation system, is needed. This generation system should be able to build all types of texture maps from different orientations, projections, and data sources.

One major source of information will be aerial photography. These photographs would be digitized, and then, using the texture map generation system, mapped onto the terrain with a perspective projection (shown in Figure 78). This projection

may occur in any orientation, depending upon the point of view from which the photograph was taken. The resolution, scale (field-of-view angle for perspective), and orientation can all be varied for total flexibility.

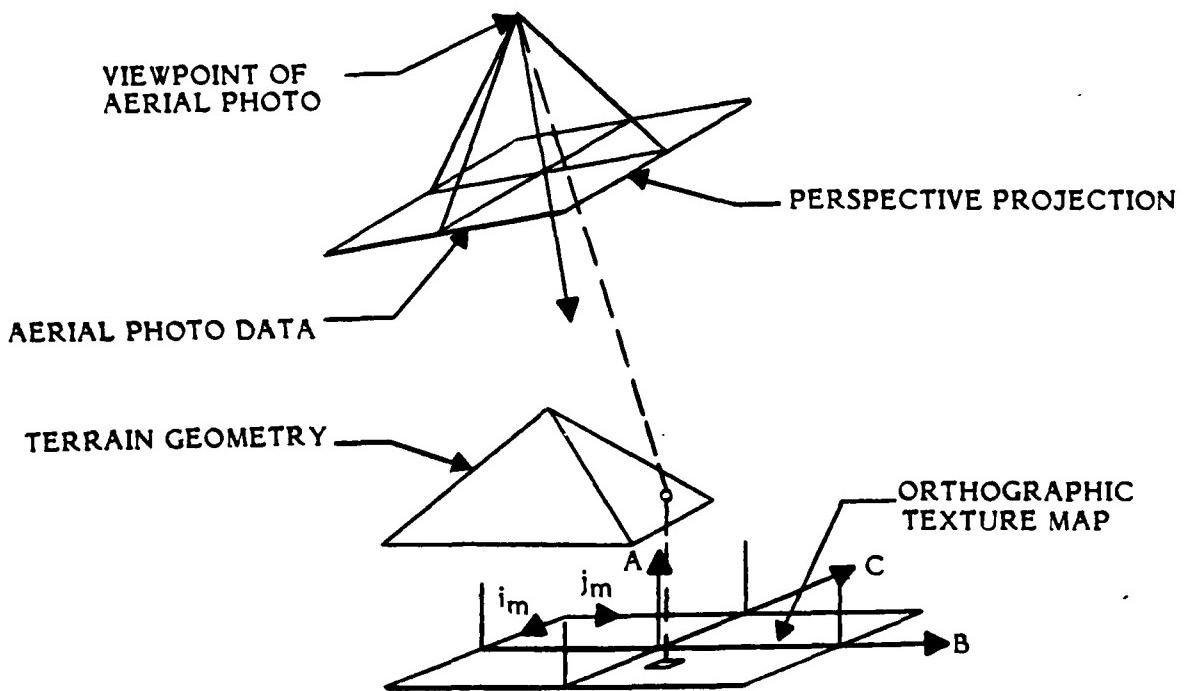


Figure 78. Projection of Aerial Photo Data to Texture

If the source for the texture map were a digitized map, an orthographic projection would be used to build the texture map. Again, scale, resolution, and orientation can all be varied to align the map with the geometry data.

Another data source is the Defense Mapping Agency's Digital Feature Attribute Data (DFAD). They consist essentially of digitized map information; a texture map could be reconstructed from these files and applied to the corresponding Digital Terrain Elevation Data (DTED). This would provide a fully automated database construction system.

As a final example of the potential of such a system, imagine projecting a photograph onto models of cultural features, such as buildings, bridges, or vehicles. The technique used would be identical to that used for aerial photography

(Figure 78). In this case, however, a number of texture maps, corresponding to the different faces of the model, would be extracted. During the image generation process, these texture maps can then be reapplied to the model (Figure 79) to provide accurate surface detail.

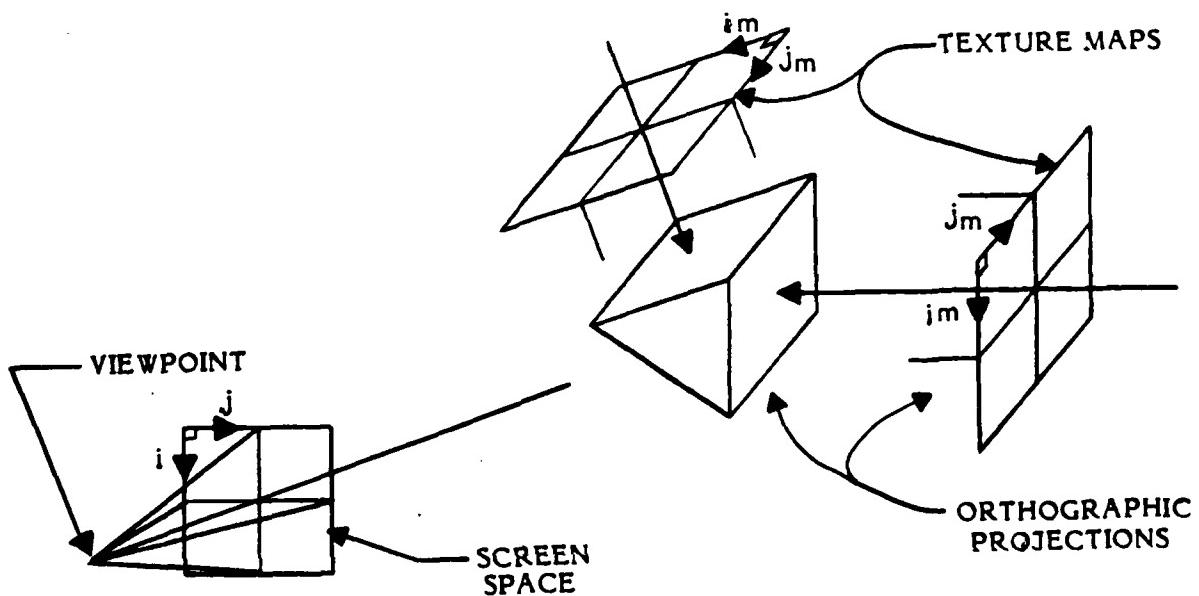


Figure 79. Projection of Texture Maps to Model Faces

In conclusion, the texture mapping technique is considered to have great potential. It is sufficiently flexible to allow arbitrary surface textures to be applied to any surface. Surface textures can be easily constructed from a wide variety of sources, and this texturing technique adds visual content to the scene without additional geometric processing.

6.5 LEVEL-OF-DETAIL CONTROL

The purpose of level-of-detail (LOD) control is to prevent processing of unnecessary detail for distant objects. The basis of LOD control is the selective use of multiple downsampled versions of data for terrain, models, and color maps. Use of appropriately downsampled data allows the computer to only process that amount of data needed to adequately portray a model or terrain. With this goal in mind, LOD control can reduce image generation time by a large percentage. Also, LOD control can act as an overload response to intelligently limit image detail for a constant throughput system.

The required criteria and objectives for this design are:

- A. To reduce unnecessary data processing to portray terrain, models, and color.
- B. To provide adequate representation of terrain and models.
- C. To have visually acceptable frame-to-frame coherence with motion over terrain.
- D. To prevent harmful errors in perception of depth or size.

6.5.1 TERRAIN LEVEL OF DETAIL

The current level-of-detail control for terrain uses a dynamic downsampling technique to reduce data for lower LOD. Real-time downsampling of data is used for simplicity. The alternative of storing predownscaled data has the disadvantage of requiring a large amount of memory to store all levels of terrain. The highest resolution base data currently resides at the leaf node level of the terrain node tree.

The LOD value for a terrain object is assigned by measuring the distance from the viewpoint to the terrain object centroid. This range is then matched with a predetermined set of boundary ranges to determine the necessary data resolution. With this LOD value, the base data is directly reduced by some percentage. There are specific requirements for dynamic downsampling of terrain data:

- A. The data reduction must be gradual for each level to provide smooth transitions.
- B. The reduced data set must be a subset of the base data versus a resampled set of the base data.
- C. The reduction must result in constant spacing between data points along either the x or y axis.

The techniques for data reduction are dependent on these requirements and the actual structure of the base data.

The current downsampling technique reduces every other data point in one axis or the other per each LOD value. This reduction provides a gradual 40% reduction in data per LOD value as shown in Figure 80.

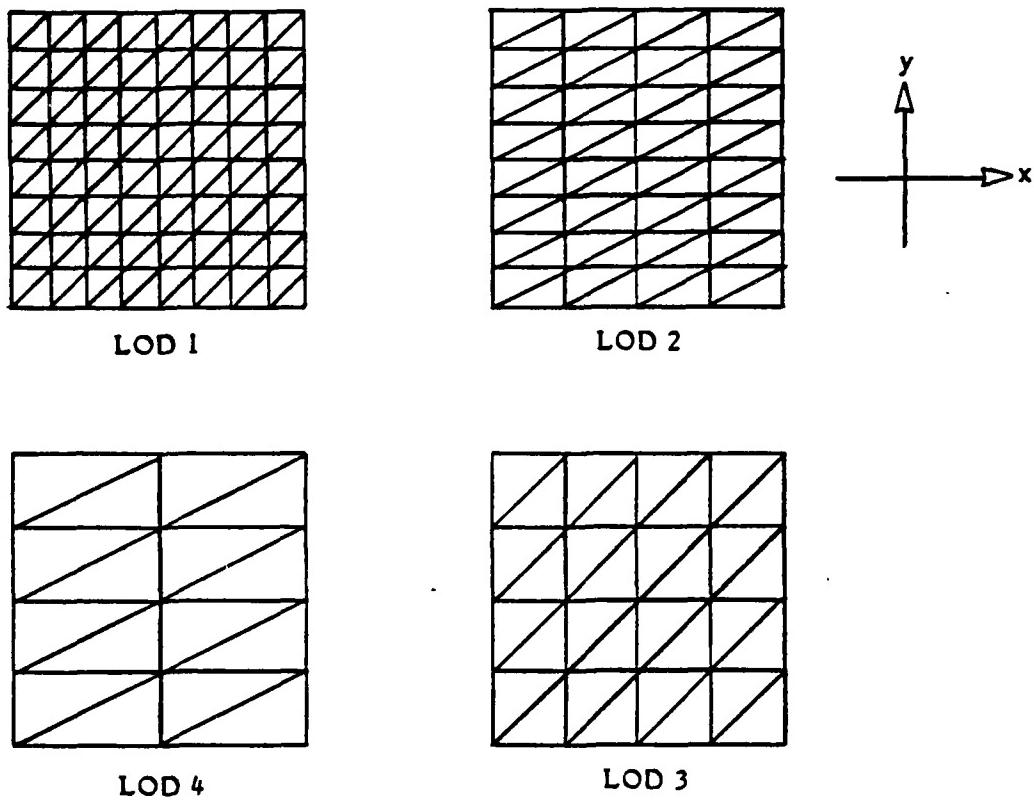
6.5.2 MODEL LEVEL OF DETAIL

The defining tree structure for model data acts as a level-of-detail control. The tree structure would define the model in many levels of detail (LOD). This is done by adding data nodes to a base model to show increased realism as seen in Figure 81.

The distance to the model centroid from the viewpoint determines how far the scheduler traverses the tree. The scheduler decides which node or groups of nodes should be used to define the model. Each node carries a predetermined LOD range value based on its data detail. The LOD ranges overlap slightly at the boundaries to reduce flicker as a model goes in and out of a LOD level. The optimum method for merging models at the LOD transition zone is still under investigation.

6.5.3 COLOR LEVEL OF DETAIL

LOD control is best applied to color by using a mip map approach as discussed in Section 6.4.



LOD	DEFINING VERTICES	% DATA REDUCTION
1	81	N/A
2	45	44
3	25	44
4	15	40
5	9	40

Figure 80. Triangle Generation of Downsampled Terrain Data

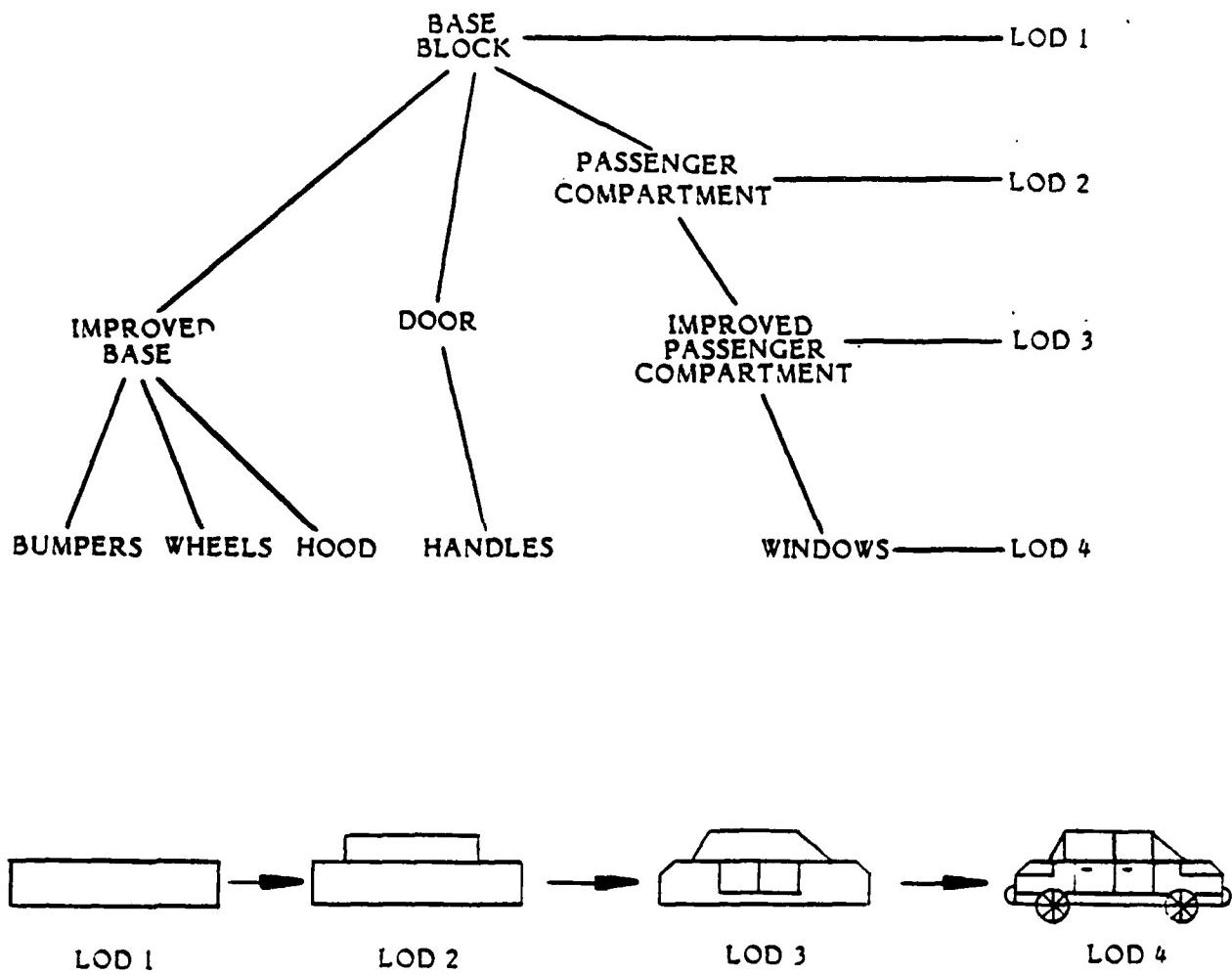


Figure 81. Model Tree with LOD

6.5.4 LOD TRANSITIONS

The transition between objects at different levels of detail is an important part of LOD control. Techniques for transitions vary for LOD control of color, models, and terrain. The problems associated with LOD transitions include cracks in the terrain model at LOD boundaries, sudden changes of realism for objects, LOD boundary flicker, and movement of terrain-based models at LOD boundaries.

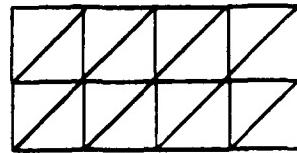
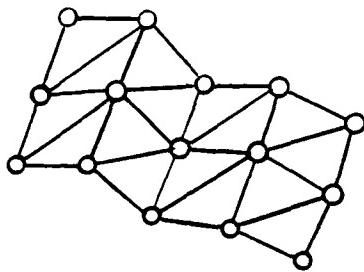
Transitions for terrain-coloring LOD are best achieved using the mip map approach. This approach uses a color value for each screen pixel as interpolated from defining color maps at different resolutions. This approach is discussed extensively in Section 6.4.

The transition between LOD's for models must be smooth. A sudden jump of realism at an LOD boundary is visually distracting. This transition is smoothed by using accurate LOD boundary ranges. A future enhancement would employ model fading through the transition. The fading is done by increasing the transparency of a model at one LOD while decreasing the transparency of the model at the adjacent LOD.

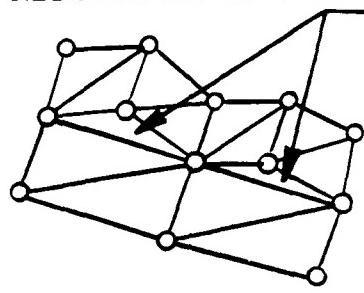
The transition for terrain objects must prevent visible changes from one LOD level to another and crack-of-the-earth. The problem of smooth transition has a dual solution. The data reduction should be as small as possible between LOD levels, since a large reduction produces very noticeable LOD waves in terrain. Also, the location of the LOD boundaries for transition must be accurately matched to the base data resolution and percentage of data reduction for each level.

The crack-of-the-earth problem involves holes in the terrain model due to a data loss at the LOD boundaries (see Figure 82). This problem may be solved in a number of ways. Extra triangles could be generated to fill in the cracks. However, the construction of such filler triangles depends on the LOD of the transition object and the surrounding objects, and is quite complicated.

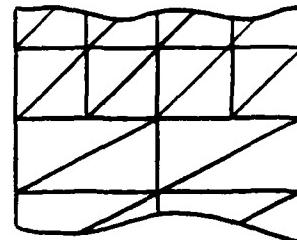
TRIANGLE GENERATION
USING ALL DATA POINTS



TRIANGLE GENERATION USING
REDUCED SET OF DATA



CRACKS DUE
TO DATA
LOSS



TERRAIN MODEL

TRIANGLE GENERATION

Figure 82. Crack-of-the-Earth Due to Data Reduction

Another solution is to process transition objects at both LOD's. This is the simplest solution, and is, in fact, the one implemented in the CIG software. The transition could be further smoothed by using the transparency fade, thereby processing terrain LOD transitions in the same way as model LOD transitions. The disadvantage of this method is that the double processing of the transition objects increases the computational load on the system.

The most elegant solution is one suggested by Lance Williams (Will83). He suggested using a mip map containing altitudes rather than colors (see Section 6.4 for a discussion on mip maps), thereby providing totally smooth transitions and no cracks. This method is, however, very computationally intense, since it would involve a bilinear and a linear interpolation for each elevation value.

6.5.5 LOD BOUNDARY RANGE

The LOD boundary ranges are the distances from the view point where a model object or terrain object would be defined using a different level of detail (see Figure 83).

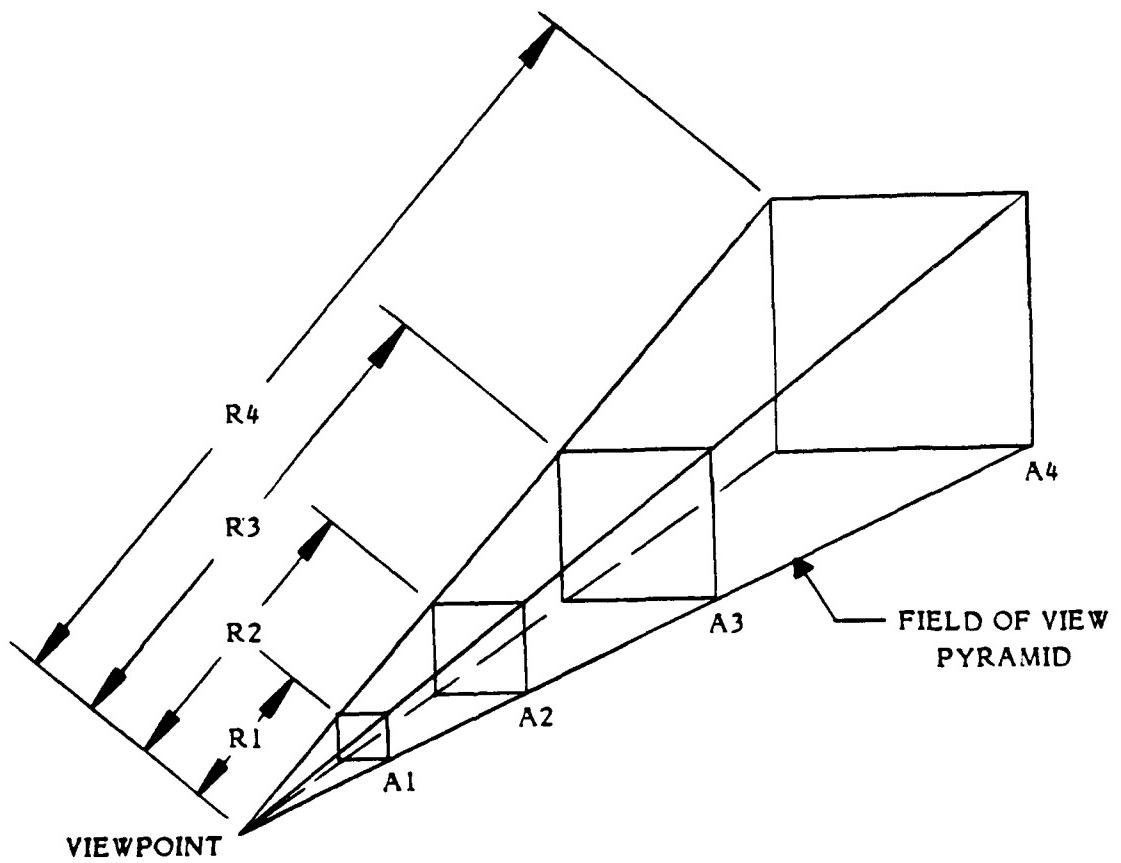


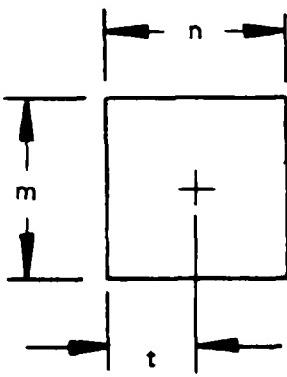
Figure 83. LOD Boundary Range Values (R_i)

At any given level-of-detail reduction, the LOD boundary range value should be at that distance such that the spacing between consecutive data points when subtended to the screen are approximately equal.

The parameters used to develop the boundary ranges include: 1) field-of-view angle, 2) base data resolution of object, 3) percentage of data reduction for each LOD, 4) surface roughness, and 5) human factors priorities. The range algorithm is developed in Table 18.

Table 18. Level of Detail Range Algorithm

R_i	: Boundary range value to LOD transition
A_i	: Area of plane \perp to FOV pyramid at range R_n
n, m	: sides of plane defining A_n
t	: 1/2 larger of n or m
Θ	: field of view angle defining pyramid
PR	: % data reduction per LOD level
Area = $n \times m$	
A_1	= base area at Range LOD 1
	= $2t \times 2 = 4t$
t_1	= $R_1 \tan(\Theta/2)$
A_i	= Area used to define LOD (i) based on % data reduction
A_i	= $\frac{1}{(1-PR)} * A_{(i-1)}$
R_i	= $A_i \frac{1}{4 * \tan(1/2\Theta)}$



The range value algorithm acts as a base for LOD boundary ranges. The highest resolution of the data will drive the distance to the first LOD transition (R_1). For example, a high density model such as a car (.5 meter resolution) will lose detail at a closer range than an elevation model at low resolution (30 meter resolution).

The ranges developed by the algorithm for terrain should represent maximum ranges for LOD transitions. The ranges could be shortened based on terrain surface roughness or human factors priorities. Surface roughness can be defined as:

$$SR = \frac{1}{\# \text{ triangles } (k)} * \sum_{n=1}^k \left(Z \text{ component of unitized face-normal of triangle.} \right)$$

This factor can be applied to LOD ranges as a future enhancement to more accurately set the ranges. For example, the plains of Kansas could be adequately represented with less detail than Colorado mountains. Human factor priorities can be waived for special circumstances to accept abrupt changes in detail in exchange for faster processing.

6.5.6 DESIGN ALTERNATIVES

The storage of terrain data at all levels of detail would eliminate the overhead required to downsample data during processing. Attaching data to higher levels of the node tree allows the scheduler to use the node tree to assist in LOD control.

This approach has great potential, but was not used at this time. The reasons for this decision include:

- A. The requirement for memory to store the terrain at the various LOD's is excessive for the current system.
- B. Data at high levels of the node tree would have greatly reduced data resolution, which has no application at this time.

7.0 PERFORMANCE EVALUATION

The architecture research system was designed as a research tool that would permit architectural concepts and algorithms to be implemented, analyzed, refined, and tested. This research culminated in an integrated system that met design specifications, employed current technology, and provided a firm foundation for future research and implementations. To prove the feasibility of the final design, the performance of this optimized system had to be evaluated. This section describes the results of the exhaustive performance tests conducted on the ARS and the conclusions drawn from these results.

7.1 HARDWARE PERFORMANCE

The special-purpose hardware of the ARS was designed as a research tool to implement and analyze a multiprocessor architecture based on the z-buffer algorithm. To facilitate the performance testing and evaluation, software on the SEL minicomputer that regulates the number and average size of triangles passed to the hardware is executed. These triangles are then processed by the custom hardware while the control/status board, using special-purpose internal counters, gathers statistics relevant to this architecture. The control/status board communicates this data to the SEL for analysis over a bidirectional HSD interface. This section presents the results of the performance tests and evaluates the hardware design on the basis of these results.

7.1.1 MEMORY MODULE PERFORMANCE

Speed was an important criterion in the development of the memory control logic. To achieve a faster implementation, memory design was based on seven cycles that took advantage of shortcuts inherent to the z-buffer algorithm. These variable-length cycles were chosen so that average memory speed would not be degraded by variations in depth complexity. The memory performance tests were selected to analyze the effectiveness of this approach.

7.1.1.1 MEASUREMENTS

The control logic section of the memory design incorporates four functions. These functions implement the basic memory write, the initialization bit test, the exponent comparison test, and the full-depth comparison test. Of the seven cycles shown in Table 19, four actually write data into memory. The background cycle which disables all functions except the memory write yields the shortest cycle. The initialization cycle is slightly longer because the initialization bit must be tested before a memory write can be performed. The look-ahead write and read-compare write both enable the comparison tests. Comparing the exponents of the pixel depths is less time-consuming than comparing the full depth values. Therefore, the read-compare write cycle is the longest. The other two write cycles do not actually access memory. The look-ahead no-write and read-compare no-write cycles are essentially look-ahead write and read-compare write cycles that

Table 19. Memory Module Cycles

CYCLE NAME	DESCRIPTION	MEASURED CYCLE LENGTH*
BACKGROUND	DISABLES CONTROL AND FORCES AN AUTOMATIC OVERWRITE.	115 ns
INITIALIZATION	FIRST WRITE CYCLE OF A NEW FRAME.	165 ns
LOOK-AHEAD NO-WRITE	EXPONENT COMPARE. NEW PIXEL DOES NOT OVERWRITE PREVIOUS PIXEL.	150 ns
LOOK-AHEAD WRITE	EXPONENT COMPARE. NEW PIXEL DOES OVERWRITE PREVIOUS PIXEL.	190 ns
READ-COMPARE NO-WRITE	FULL DEPTH COMPARE. NEW PIXEL DOES NOT OVERWRITE PREVIOUS PIXEL.	170 ns
READ-COMPARE WRITE	FULL DEPTH COMPARE. NEW PIXEL DOES OVERWRITE PREVIOUS PIXEL.	210 ns
READ	READ MEMORY CONTENTS.	240 ns

*ns = nanosecond = 1×10^{-9} second

terminate before the memory write. This occurs when the comparison test indicates that the new pixel is obscured by the previous pixel. The read cycle simply transfers the contents of the memories to the antialiaser.

An example of the memory statistics gathered by the control/status board is shown in Table 20. The total accesses of each memory module are listed under the CYCLES column. The other columns represent the cycle count for each of the six memory write/no-write cycles. Also tabulated are the calculated depth complexities for each memory module and the system. This breakdown by cycle provides the data required to analyze variations in memory cycle utilization.

7.1.1.2 EVALUATIONS

The performance evaluation tests were designed to verify the effectiveness of the variable cycle approach in minimizing the average cycle length. Two thousand triangles were tiled in each test with the distribution tree operating in scramble mode to ensure an even distribution of accesses among the eight memory modules. The depth complexity was varied by controlling the average triangle size. The background cycle was eliminated from the testing because it simply provides an overwrite capability and is not unique to the z-buffer approach.

The bar graphs depicted in Figure 84 display the percentages of each cycle utilized in processing a specific depth complexity. At a low depth complexity, the initialization cycle dominates the determination of the average cycle length. As the depth complexity increases, the effects of the shorter look-ahead no-write and read-compare no-write cycles become increasingly important. These factors are more dramatically illustrated by the curves in Figure 85. Design alternative 1 represents the memory module design implemented in the ARS hardware, and shows the effect variable-length cycles have on maintaining a uniform average cycle length. The other three curves represent alternatives to this design that sacrifice speed for a simpler control structure. For example, the control logic could be simplified by eliminating the circuitry that terminates the look-ahead write and read-compare write cycles when a no-write cycle is required. This would replace the four look-ahead and read-compare cycles with look-ahead write no-write and read-compare write no-write cycles. This example is illustrated by alternative 2. Alternative 3 further reduces the control logic by eliminating the

Table 20. Memory Statistics

Breakdown of the number of different memory cycles utilized in a typical test image.

MEMORY NUMBER	CYCLES	INITIALIZE WRITE	BACKGROUND WRITE	LOOK AHEAD WRITE	NO WRITE	READ COMPARE WRITE	READ COMPARE WRITE	DEPTH COMPLEXITY
0	30377	19603	0	3156	3970	1750	1898	.927
1	30430	19606	0	3208	3949	1763	1904	.928
2	30393	19362	0	3157	4122	1696	2056	.927
3	30620	19755	0	3250	3990	1749	1876	.934
4	30411	19619	0	3214	3934	1777	1867	.928
5	30416	19647	0	3154	3905	1813	1897	.928
6	30502	19737	0	3156	3953	1794	1862	.930
7	30458	19635	0	3204	3942	1772	1905	.929

TOTAL MEMORY CYCLES = 243607

SYSTEM DEPTH COMPLEXITY = 0.929

TRIANGLES TILED = 2000

AVERAGE TRIANGLE SIZE = 122 PIXELS

DISTRIBUTION TREE MODE = SCRAMBLE

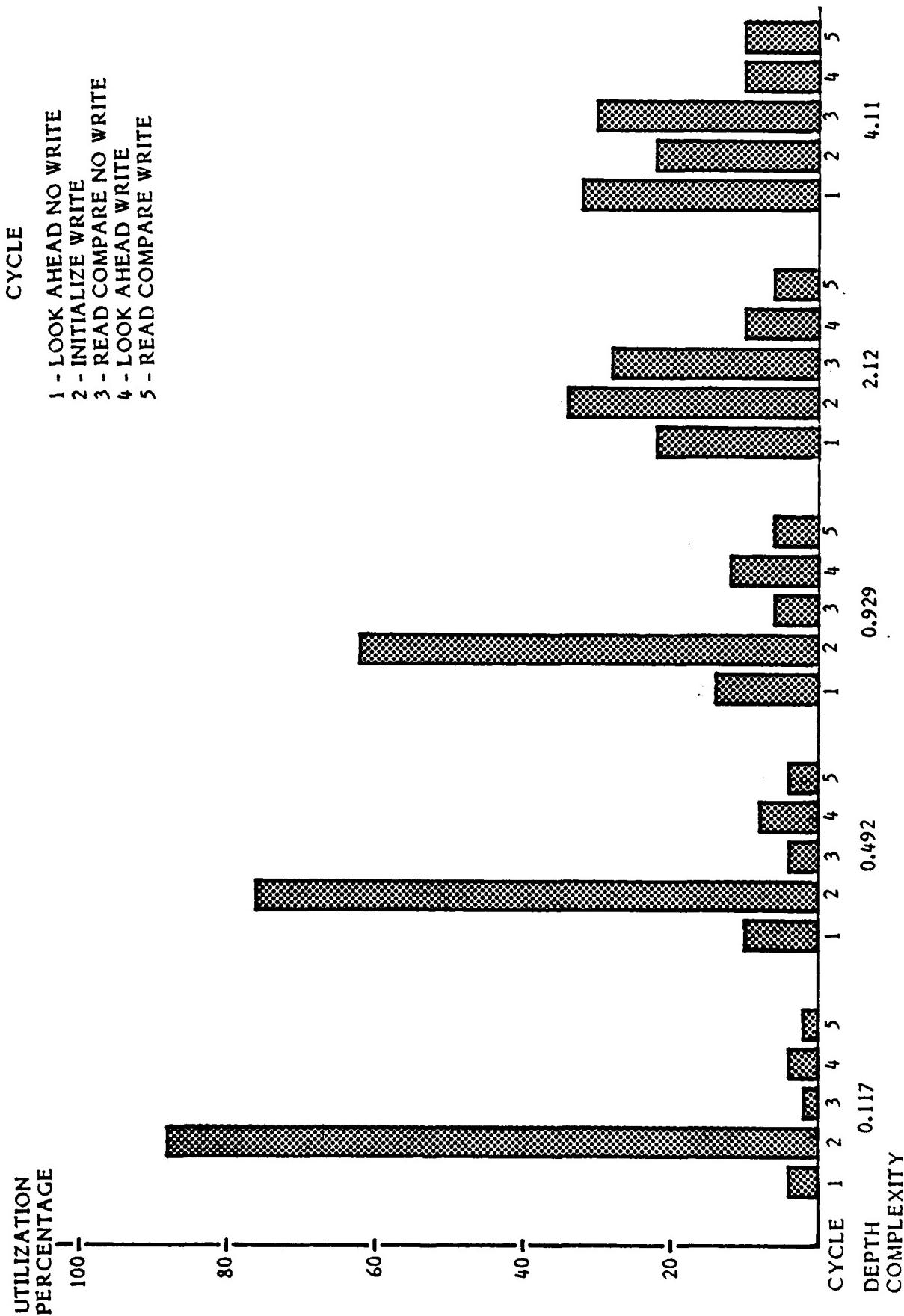


Figure • Memory Module Cycle Utilization

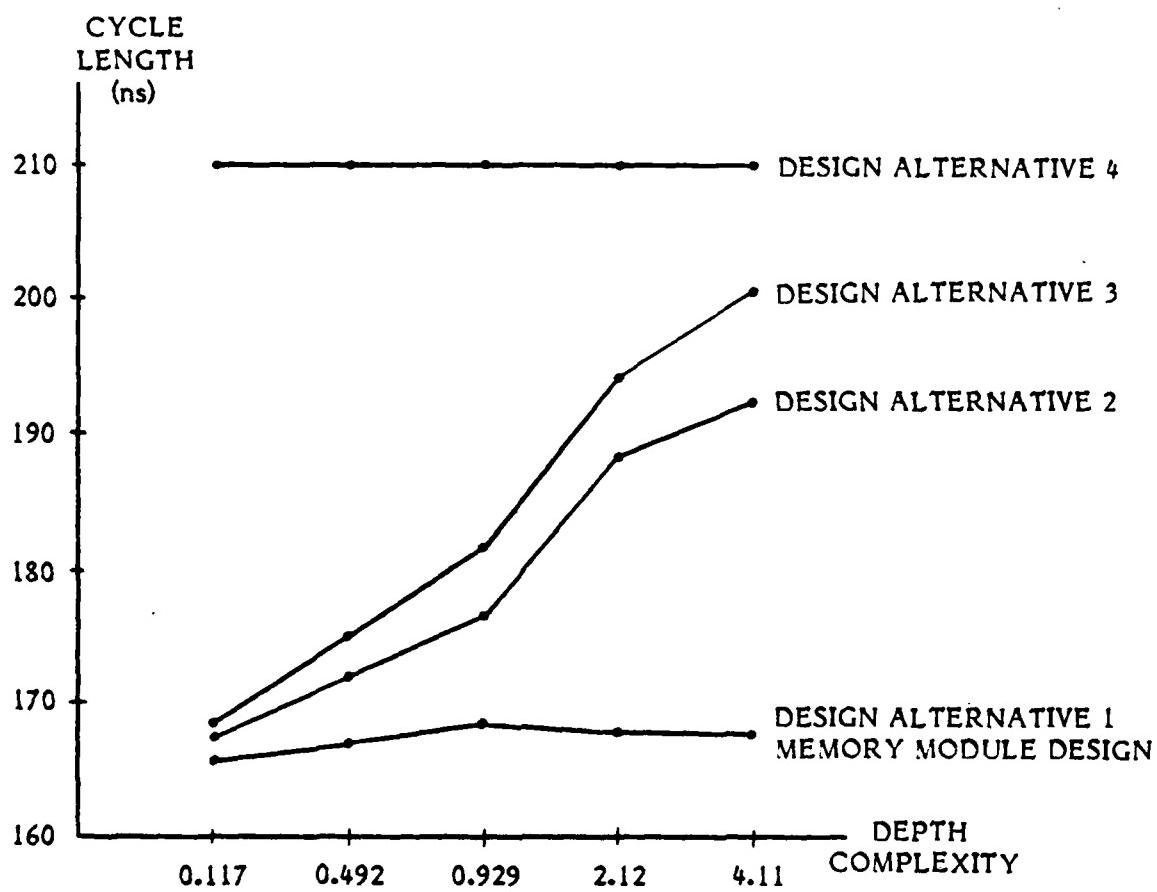


Figure 85. Average Memory Cycle Lengths for Four Design Alternatives

time-saving exponent compare function resulting in a design utilizing only the initialization and read-compare write no-write cycles. Finally, a design utilizing a single fixed-length cycle yields the results for alternative 4.

The memory module performance evaluation clearly shows the benefits of a variable-length cycle design for increasing speed and minimizing average cycle lengths. Although the control logic is more complex, the ARS memory module design reduces the average cycle length by 20% over a design implementing a single fixed-length cycle.

7.1.2 TILER PERFORMANCE

The rates at which triangles are tiled and pixels are produced were identified as important performance parameters during development of the tiler architecture. To achieve the design goals of 100,000 triangles per second and 10 million pixels per second, the tiler was implemented as a multiple-stage, parallel-processing computer. Two of these stages, the tiling machine setup and the tiling machine, implement triangle-rendering algorithms and, therefore, dominate tiler performance. The performance tests for the tiler were designed primarily to analyze the efficiency of the rendering algorithms and their impact on the tiling and pixel output rates.

7.1.2.1 MEASUREMENTS

The tiler architecture incorporates four functional stages. These stages are the HSD interface, the tiling machine setup, the tiling machine, and the tile accumulate. Since the tiler is a multiple-stage or pipelined computer, it exhibits transport delay. This is simply the sum of the average delays at each processing stage. It is an indication of the amount of time required for data to traverse the stages. The critical stages of the tiler, their associated delays, and the total transport delay are illustrated in Figure 86.

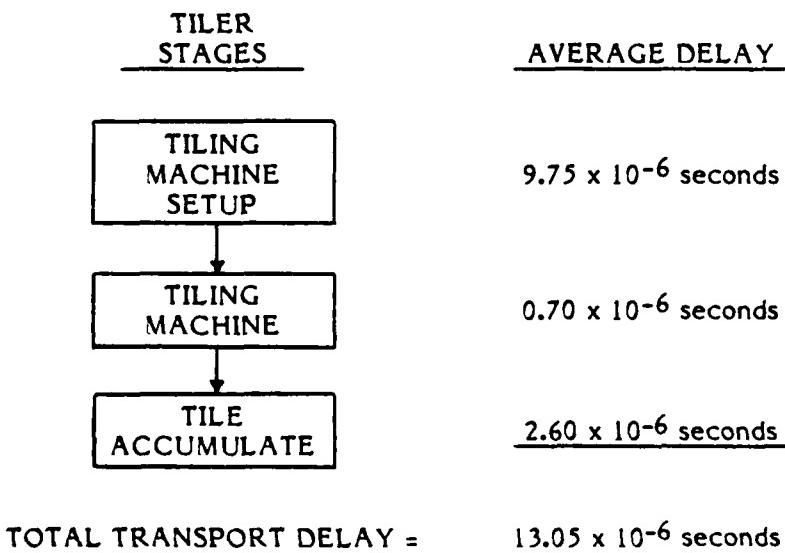


Figure 86. Tiler Transport Delay

Separating the tiling process into two stages, the tiling machine setup and the tiling machine, is an important concept in achieving high throughput. This concept allows concurrent processing of two triangles; one triangle is tiled while setup calculations for the next triangle are being performed. The design goal of 100,000 triangles per second translates to a processing time of 10 microseconds per triangle. This is possible if each stage performs its tasks in 10 microseconds or less. The time required for the tiling machine setup stage to execute its task varies slightly depending on triangle orientation and the sampling mode selected. Therefore, this stage was designed to perform all setup calculations in 10 microseconds or less. The actual processing times for both modes are listed in Table 21 below.

Table 21. Tiling Machine Setup Average Processing Time per Triangle

UNDERSAMPLE	OVERSAMPLE
9.75 microseconds	8.63 microseconds

These average values were obtained by using the busy/not busy status flag to determine total processing time for two thousand random triangles.

Once the setup calculations are completed, the data is transferred to the tiling machine where high-speed rendering or tiling is performed. The design goal of 10 million pixels per second translates to a tiling rate of 100 microseconds per pixel. Therefore, the tiling machine was designed to sample a pixel every 100 microseconds. Sampling a pixel does not necessarily mean that the pixel will be subsequently tiled. The effects of this inefficiency will be discussed in the next subsection. Since the tiling machine operates on pixels, the processing time is based on pixels per triangle. Figure 87 illustrates the effect of increased triangle size on tiler throughput rates. The triangle throughput rate is constant at 100,000 triangles per second up to a triangle size of 100 pixels. At this point, the processing times of both stages are equal (i.e. 100 pixels \times 100 ns per pixel = 10 microseconds), and triangle throughput and pixel output rates are optimized. As triangle size expands beyond 100 pixels, the tiling machine determines triangle throughput rates.

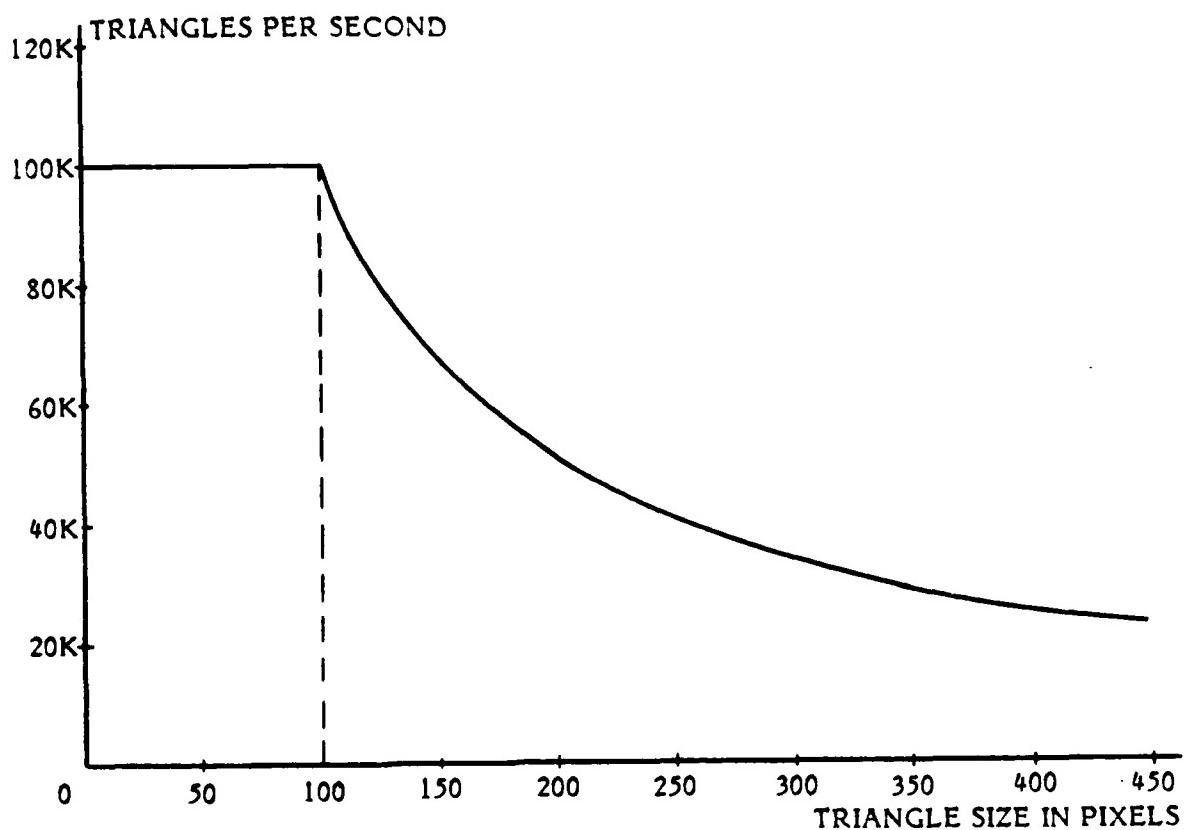


Figure 87. Effect of Triangle Size on Tiler Throughput Rates

A detailed discussion of the tiling algorithm implemented in the ARS tiler is presented in subsection 4.2.2. Part of that discussion focuses on sampling paths and sampling efficiency. Since sampling efficiency can significantly affect tiler performance, the control/status board was configured to gather data necessary to calculate average sampling efficiency. An example of this data is provided in Table 22. The number of triangles, approximate triangle size, and sampling mode are specified. During tiling, the control/status board counts the pixels sampled and the pixels tiled for each tiler. This data is then passed to the SEL where the averages are computed. The results of sampling efficiency measurements for a range of triangle sizes are tabulated in Table 23. Another method of analyzing efficiency is to measure the pixel output rate. This approach measures tiling efficiency, and incorporates both sampling efficiency and tiling machine delays. The results of tiling efficiency measurements are tabulated in Table 24. The data was generated by controlling triangle area and using the tiling machine busy/not

Table 22. Tiler Statistics
 Breakdown of tiler sampling efficiency for a typical test image

TILER NUMBER	TRIANGLES TILED	PIXELS SAMPLED	PIXELS TILED	AVERAGE TRIANGLE SIZE IN PIXELS	AVERAGE SAMPLING EFFICIENCY (%)
1	500	86584	61086	122.172	70.551
2	500	88473	61770	123.540	69.818
3	500	83547	59815	119.630	71.594
4	500	86436	60935	121.870	70.497
TOTAL	2000	345040	243606	121.803	70.602

SPECIFY
 TOTAL TRIANGLES TILED = 2000
 APPROXIMATE TRIANGLE SIZE IN PIXELS = 100
 OVERSAMPLE/UNDERSAMPLE MODE = OVERRSAMPLE

Table 23. Sampling Efficiencies for Undersample and Oversample Modes

AVERAGE TRIANGLE SIZE IN PIXELS	UNDERSAMPLE		OVERSAMPLE	
	AVERAGE SAMPLING EFFICIENCY (%)	AVERAGE TRIANGLE SIZE IN PIXELS	AVERAGE SAMPLING EFFICIENCY (%)	AVERAGE SAMPLING EFFICIENCY (%)
8.6	23.5	15.2	15.2	60.6
32.5	15.6	52.7	52.7	65.3
49.7	14.6	75.9	75.9	67.6
85.7	15.1	121.5	121.5	71.7
120.4	10.5	166.7	166.7	72.3
157.2	11.1	210.8	210.8	74.3
202.8	55.7	247.4	247.4	81.0
240.8	57.4	289.8	289.8	81.9
279.7	55.7	332.2	332.2	82.9
374.8	60.5	435.2	435.2	84.7
473.2	68.3	541.0	541.0	86.0

Table 24. Tiling Efficiencies for Undersample and Oversample Modes

AVERAGE TRIANGLE SIZE IN PIXELS	UNDERSAMPLE			OVERSAMPLE		
	MEASURED PIXELS/SEC	TILING EFFICIENCY (%)	AVERAGE TRIANGLE SIZE IN PIXELS	MEASURED PIXELS/SEC	TILING EFFICIENCY (%)	
8.6	0.91×10^6	-	15.4	1.87×10^6	-	-
32.6	1.41×10^6	14.1	52.7	4.48×10^6	-	-
49.7	1.40×10^6	14.0	75.8	5.74×10^6	-	-
85.0	1.26×10^6	12.6	121.6	6.83×10^6	68.3	
121.3	1.31×10^6	13.1	165.9	7.15×10^6	71.5	
157.6	1.26×10^6	12.6	209.6	7.33×10^6	73.3	
202.5	5.43×10^6	54.3	247.5	7.93×10^6	79.3	
240.1	5.37×10^6	53.7	289.8	8.04×10^6	80.4	
279.5	5.33×10^6	53.3	332.6	8.15×10^6	81.5	
375.6	6.28×10^6	62.8	435.4	8.38×10^6	83.8	
473.2	6.84×10^6	68.4	540.3	8.53×10^6	85.3	

busy flag to determine total tiling time for two thousand random triangles. Pixel output rate can be determined by dividing the total number of pixels tiled by the time required to tile them. The tiling efficiency is simply the measured rate as a percentage of the 10 million pixels per second maximum rate. The processing time of the tiling machine is determined by the number of pixels sampled. If the average number of pixels sampled per triangle falls below approximately 100 pixels, then the tiler performance is determined by the tiling machine setup stage. This causes the maximum pixel output rate to drop below 10 million pixels per second. Under this condition, tiling machine efficiency as defined is meaningless and no values are given.

7.1.2.2 EVALUATIONS

The tiling machine setup stage spends an average of 8.63 microseconds on each triangle processed in the oversample mode. Since the pixel sampling time is 100ns or one cycle, optimum triangle throughput and pixel output rates result when the average tiling machine processing time is 86 cycles per triangle. Figure 86 shows a delay of 0.7×10^{-6} seconds or 7 cycles associated with the tiling machine. This delay represents the time required to load data from the tiling machine setup stage. Therefore, the 86-cycle optimum processing time can be broken down into 79 cycles for sampling pixels and 7 cycles for loading data. The design implementation curve in Figure 88 illustrates this fact. The maximum triangle throughput exceeds the design goal of 100,000 triangles per second, but begins to degrade at an average triangle size of 79 pixels. The discrepancy between the design goals and the design implementation result from the 7-cycle overhead in the tiling machine. The same analysis holds true for the undersample mode. The tiling machine setup average of 9.75 microseconds per triangle equates to a 90-cycle optimum processing time in the tiling machine. Figure 89 shows the design implementation curve for the undersample mode.

The pixel output rate of the tiler is directly related to the efficiency of its rendering algorithm. Figure 90 illustrates this fact by presenting four tiler implementations. A tiling machine that receives setup data every 10 microseconds, loads without an overhead delay and implements a rendering algorithm that is 100% efficient is presented as the ideal implementation. Modifying this ideal case to account for the actual performance of the tiling machine setup stage

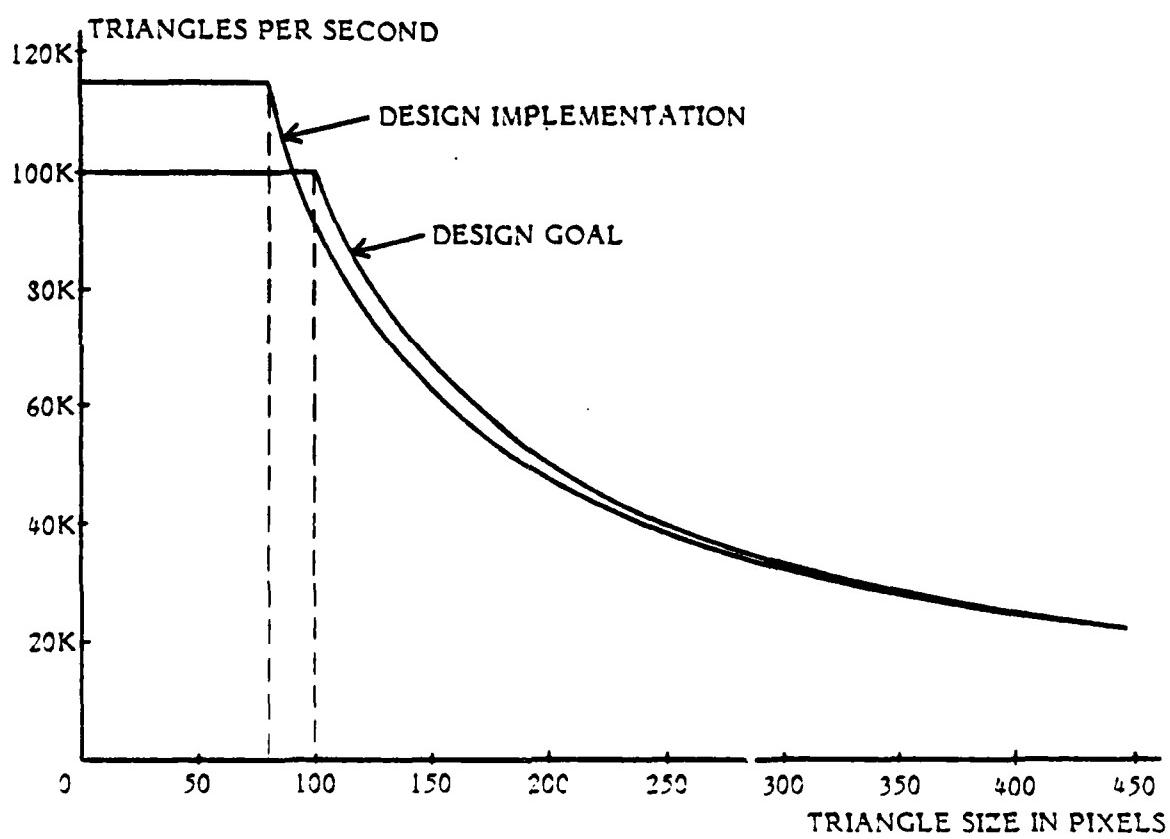


Figure 88. Tiling Machine Setup Stage Performance for Oversample Mode

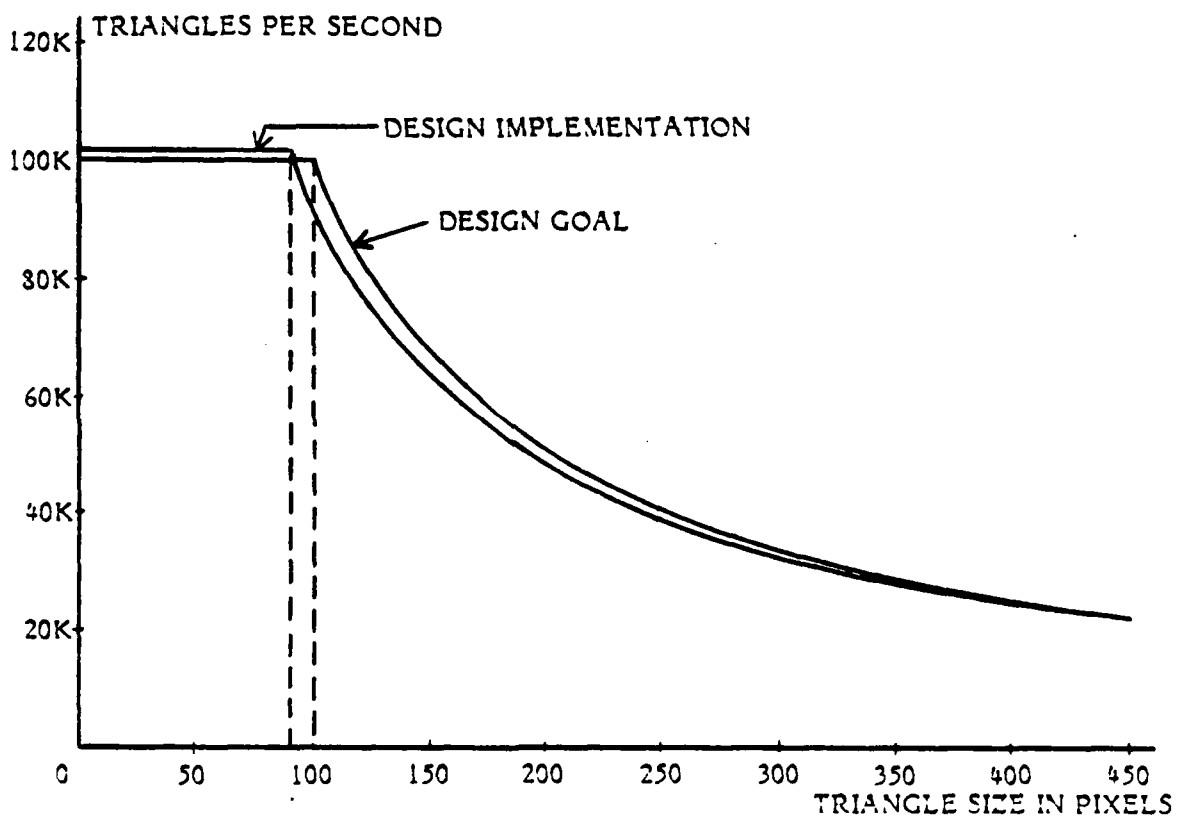


Figure 82 Tiling Machine Setup Stage Performance for Undersample Mode

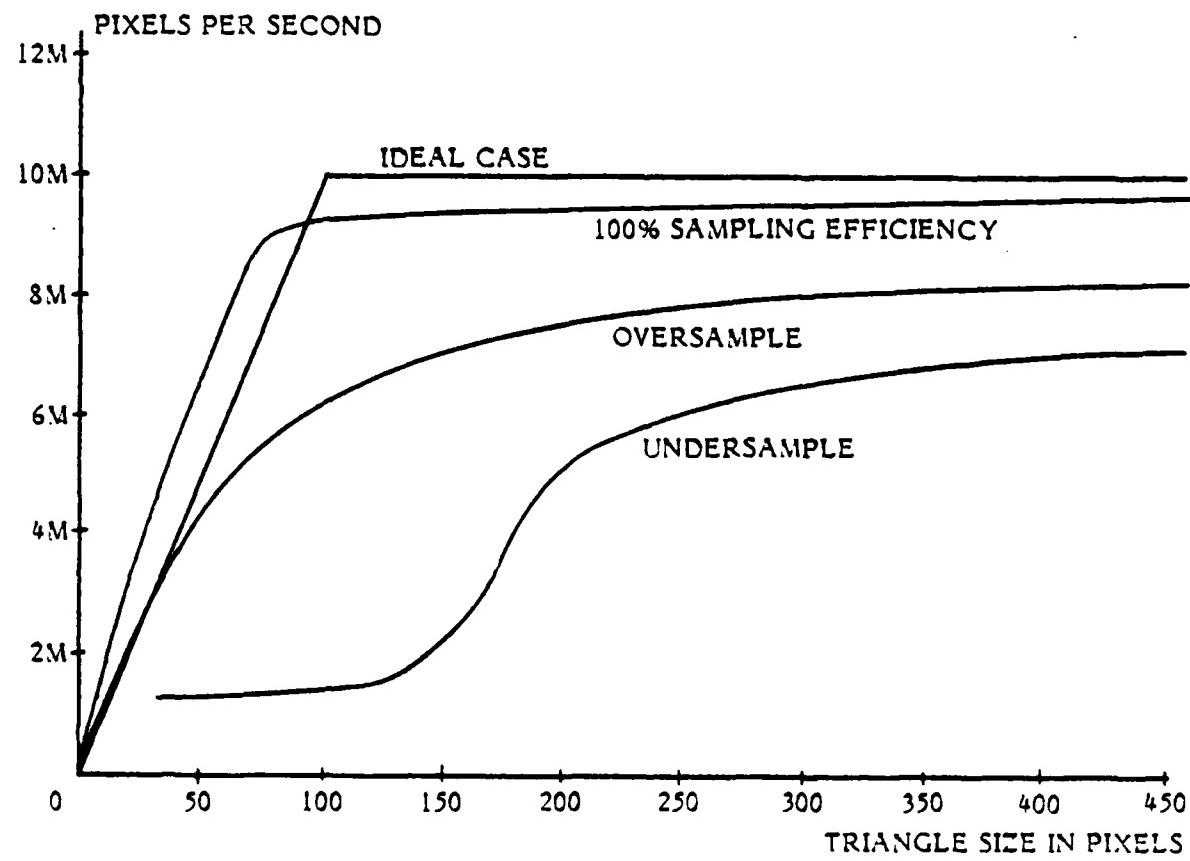


Figure 90. Tiling Efficiency for Undersample and Oversample Modes

and the seven-cycle data load overhead yields the 100% sampling efficiency curve. Using this curve as the tiling efficiency baseline for the ARS tiler, the effects of an inefficient sampling algorithm are obvious. Both undersample and oversample modes become increasingly efficient with expanding triangle size simply because the ratio of boundary pixels to interior pixels is decreasing. Another factor has an important effect on performance in the undersample mode especially when processing small triangles. Figure 12 in subsection 4.2.2.5 illustrates an under-sampled triangle where no interior pixels are tiled for several scan lines. During tiling, the tiling machine will sample every pixel along a scan line from the leading triangle edge to the edge of the bounding rectangle if no interior pixels are found. This significantly increases the ratio of pixels sampled to pixels tiled, and efficiency is reduced. This situation does not occur in the oversample mode because the tolerance was designed to ensure that a pixel would be tiled on each scan line.

The overall architecture of the ARS tiler has proven highly satisfactory. The tiling machine setup stage exceeded the maximum triangle throughput rate of 100,000 triangles per second. The tiling machine stage comes close to attaining the pixel output rate of 10 million pixels per second under most operating conditions. Additional work directed toward increasing the speed of the tiling machine and improving the sampling efficiency will significantly improve the pixel output rate.

7.1.3 DISTRIBUTION TREE PERFORMANCE

Minimizing data contention was a major concern during the design and development of the distribution tree. If data contention was successfully minimized, then queue lengths could be reduced, resulting in simplification of network circuitry and increased data throughput. To effectively reduce contention, data traffic must be uniformly distributed through the network. This allows each memory to be accessed consistently with a minimum of backlog. The distribution tree performance tests were selected to analyze contention by manipulating data traffic. Queue lengths and memory access distributions were used as a measure of contention levels.

7.1.3.1 MEASUREMENTS

To manipulate data traffic, three separate hashing functions were implemented in the control logic of the A-cells. Each hashing function employed a different scheme for altering the memory destination of a pixel. The sequential mode essentially allowed each pixel to be routed to its original memory location. The interleave mode routed adjacent pixels to adjacent memories. The scramble mode placed adjacent pixels three memories apart.

Reductions in the distribution tree queue lengths is a good indication of lower contention levels. As the tilers operate closer to their maximum rate, data traffic on the network increases, creating a higher probability of contention and impacting queue lengths adversely. The control/status board was designed to monitor all queues internal to the distribution tree and record their maximum lengths. All three hashing modes were tested with increasing data rates, and the queue lengths required to support those rates were measured. The results of these tests are presented in Figure 91. As data traffic increased, the queue lengths required for the sequential mode rose dramatically, approaching the system maximum of 256. Operation in the interleave mode allowed queue lengths to remain reasonably low and constant at all speeds. The requirement for queues in the scramble mode was almost non-existent, even with the tilers operating near their maximum rate of ten million pixels per second.

Another method of analyzing contention involves monitoring the distribution of memory accesses. The memory statistics obtained from the tests used to measure queue lengths were evaluated with this in mind. Figure 92 displays the distribution of accesses across all memories achieved by each of the three hashing modes. The sequential mode data indicates that the test image is heavily biased toward the central memories. The effectiveness of the interleave and scramble modes in removing that bias is quite evident. Providing a uniform distribution of memory accesses contributes a great deal toward increasing data throughput and minimizing contention.

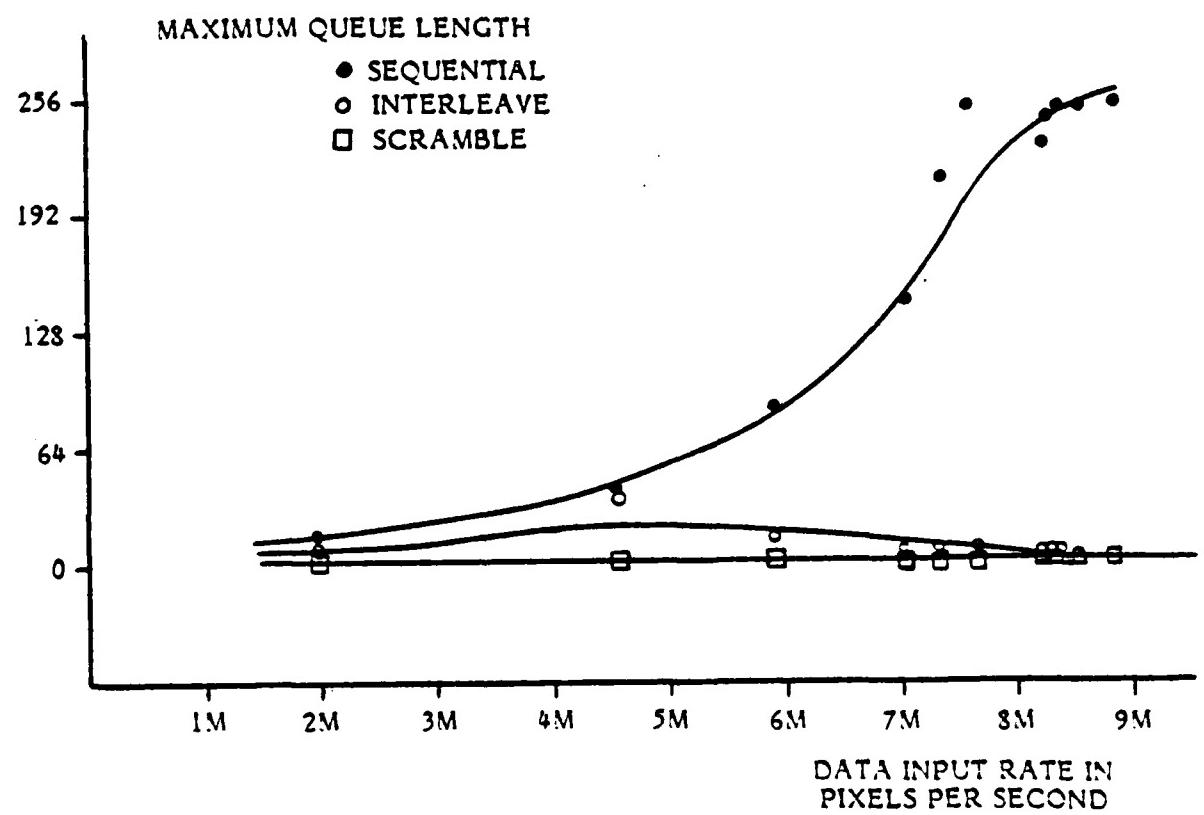


Figure 91. Maximum Queue Lengths Required for Sequential, Interleave, and Scramble Modes at Varying Data Rates

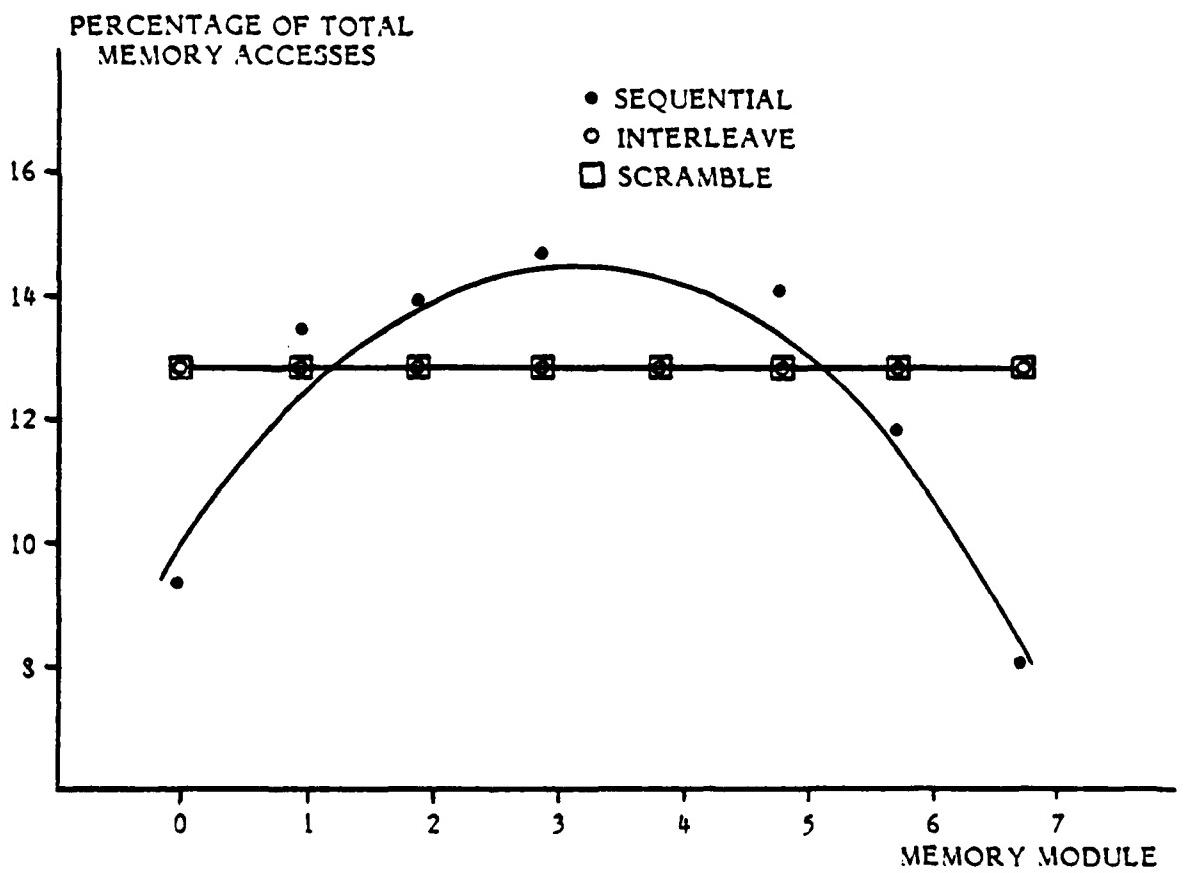


Figure 92 Distribution of Memory Accesses for Sequential, Interleave, Scramble Modes

7.1.3.2 EVALUATION

The performance testing clearly illustrated the effectiveness of the scramble and interleave modes in minimizing contention on the distribution tree. These modes achieved a uniform distribution of data traffic, decreased queue lengths to an absolute minimum, and allowed the distribution tree to achieve an input/output rate of forty million pixels per second.

Low contention rates in the network are also directly attributable to tiler efficiency. Since the tiler operates at less than its maximum rate, the network is not totally saturated, decreasing utilization and the probability of contention.

7.2 SOFTWARE PERFORMANCE

Two types of tests were used to monitor the performance of the CIG software:

- o CPU execution times of all subroutines, and counts of subroutine and entry point invocations; and
- o counts of key statistical variables.

These measurements were designed to investigate the execution frequency of key procedures in the generation of a typical CIG image, and the proportion of the total execution time that is consumed by each procedure. Such measurements permit an evaluation of current algorithms and form a foundation for additional development and optimization.

7.2.1 DESCRIPTION OF PERFORMANCE TESTS

A report was generated for each of the two types of performance tests. The CPU execution times cited in this section were measured on a Gould SEL 32/8705 under the MPX-32 operating system (version 3.2) and using the Fortran-77+ compiler (version 4.0). The SEL 32/87 is rated at approximately 3.7 million instructions per second (unoptimized Whetstone) (Mokh82). The execution times cited include the update time for the statistical counters which adds approximately 60% to the overall execution time. The timing and subroutine call counts were measured with a commercial debugging and optimization package, AID (trademark of the International Software Corporation, Golden, Co.).

The statistical variables were measured by conditionally compiled counters at key locations in the code. An analysis of this information was printed at the end of the frame or set of frames.

The scenario used to generate the measurements consisted of a simulated flight approach to an airport, and is described in detail in Table 25. This scenario was used to make actual measurements of quantities which are constant, as well as typical measurements of quantities which may vary from frame to frame.

Table 25. Scenario for Software Performance Measurement

Scene content	Approach to airport. Airport lies in a valley and includes several aircraft on taxiway.
Scene complexity	21,000 triangles used to describe terrain (no LOD control) 3,600 triangles used to describe airport and aircraft.
Field of view	32° horizontal.
Viewpoint position	100 meters altitude, 600 meters from threshold of runway (approximate)
Depth complexity	1.74

7.2.2 PRETILER PERFORMANCE

The pretiler performs four basic tasks: hither/yon clipping, screen space projection, screen clipping, and tiler set-up. It consumes the majority of the total software execution time (see Figures 93 and 94). It is a logical candidate for future hardware implementation due to its computational intensity and its position immediately preceding the tiler in the execution pipeline. Optimization of its algorithms is therefore of prime importance.

The most time-consuming task of the pretiler is the tiler set-up (see Figures 93 and 94). This task primarily involves calculating the color and depth parameters required as input to the tiler. For example, the depth parameters d_0 , $\frac{\partial d}{\partial i}$, and $\frac{\partial d}{\partial j}$ can be calculated from the original vertex data by solving a set of linear equations:

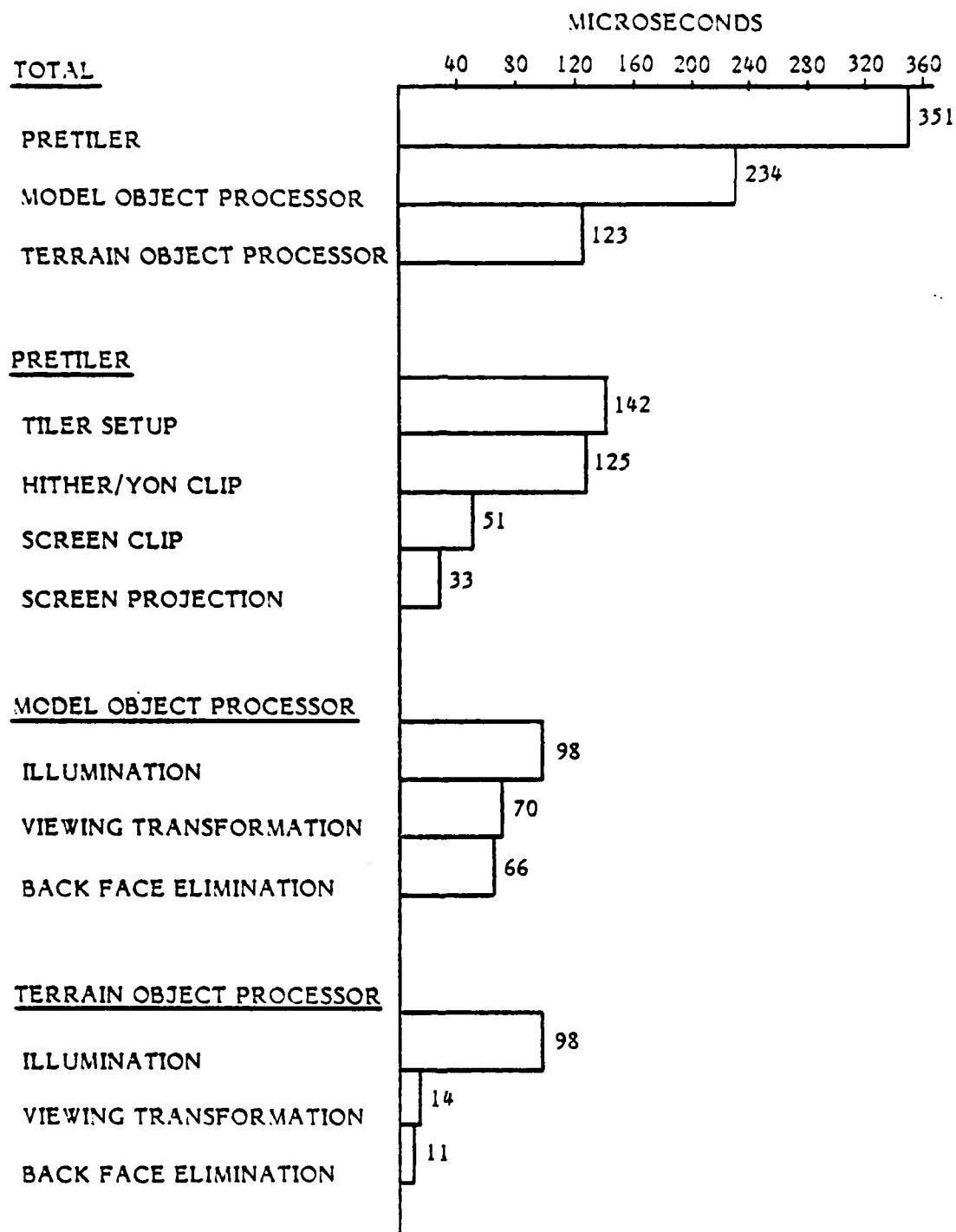


Figure 93. Relative Time per Face for Major Software Tasks

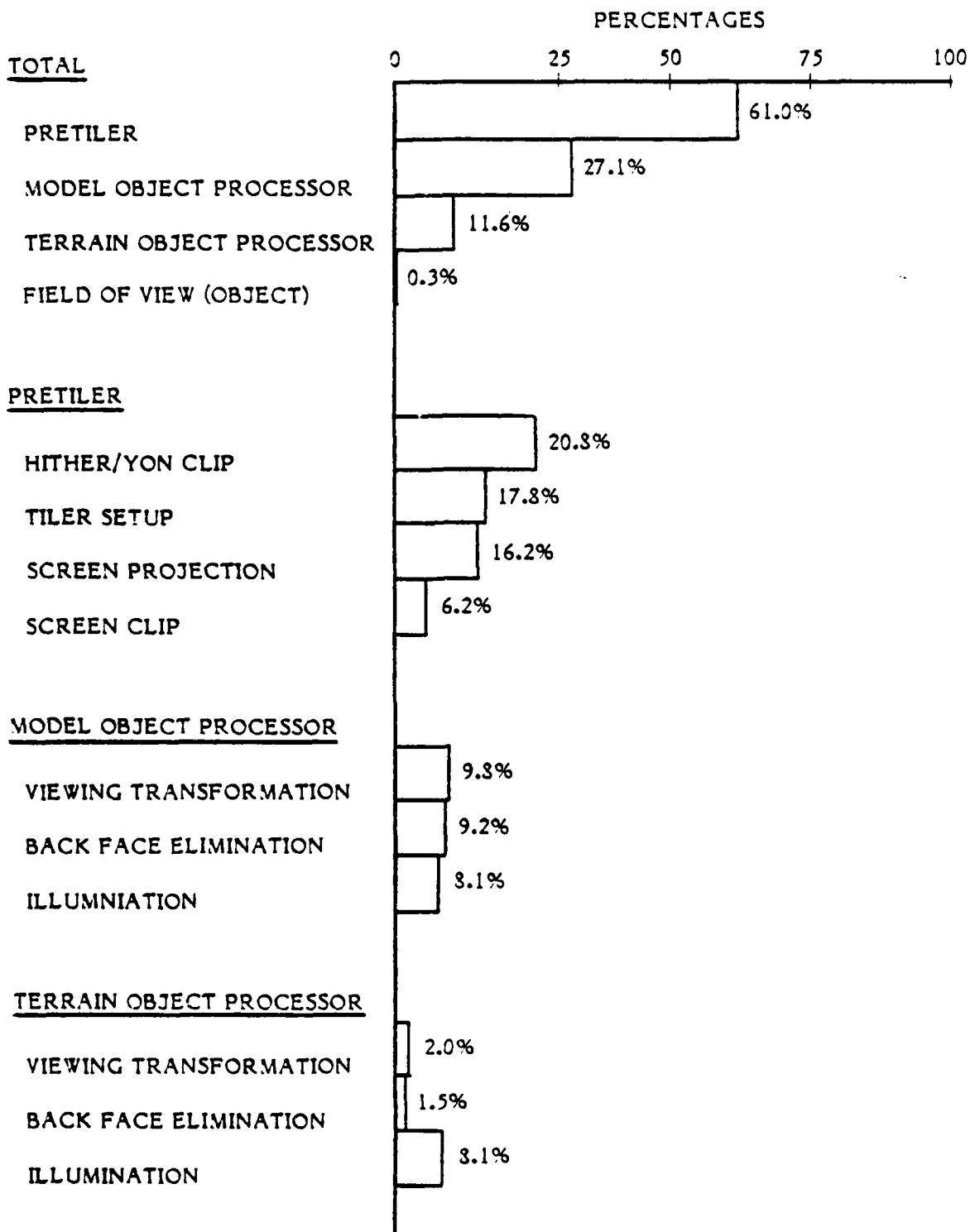
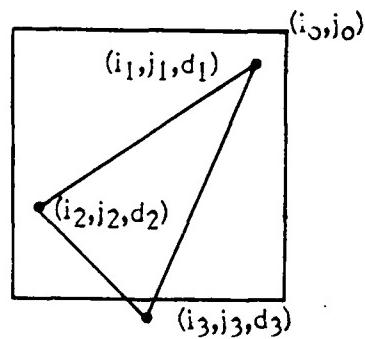


Figure 94. Percentage of Time per Frame for Major Tasks (Typical Frame)

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 1 & \Delta i_1 & \Delta j_1 \\ 1 & \Delta i_2 & \Delta j_2 \\ 1 & \Delta i_3 & \Delta j_3 \end{bmatrix} \cdot \begin{bmatrix} d_0 \\ \frac{\partial d}{\partial i} \\ \frac{\partial d}{\partial j} \end{bmatrix}$$



where (i_0, j_0) is the origin of the triangle (rounded to the nearest pixel);
 $(\Delta i_x, \Delta j_x) = (i_x - i_0, j_x - j_0)$, $x=1,2,3$ are the displacements of the actual vertices from the origin; and
 d_1, d_2, d_3 are the actual depths at the vertices.

Determining the depth and color parameters directly from this form, however, leads to precision problems involving the calculations of Δi and Δj . In the example shown, (i_1, j_1) and (i_0, j_0) are very nearly equal. Therefore, due to limited floating-point precision, $(\Delta i_1, \Delta j_1) = (i_1 - i_0, j_1 - j_0)$ may contain substantial error for large (i_0, j_0) .

Hence, it was necessary to use an alternate, more complicated method to calculate the depth and color parameters. An intense effort to simplify the calculation without precision degradation did not produce significant results, and the tiler set-up continues to be the slowest task of the pretiler.

The next slowest pretiler task is hither/yon clipping. In reality, triangles are only clipped to the hither plane, and are simply rejected if they lie totally beyond the yon plane. In view of this simplification and the infrequency of actual hither clipping (see Figure 95), the large amount of time consumed by the hither/yon clipping is surprising. This fact is primarily attributable to the trivial accept/reject tests that are applied to every face. This task requires further optimization to increase system throughput.

Screen clipping is one of the two fastest tasks of the pretiler. Screen clipping also has a very low incidence of occurrence. Nonetheless, the clipping algorithm can be substantially simplified by readjusting the tiler's circumscribing rectangle (see Section 5.2 and Figure 52). Such simplifications have little impact in terms of

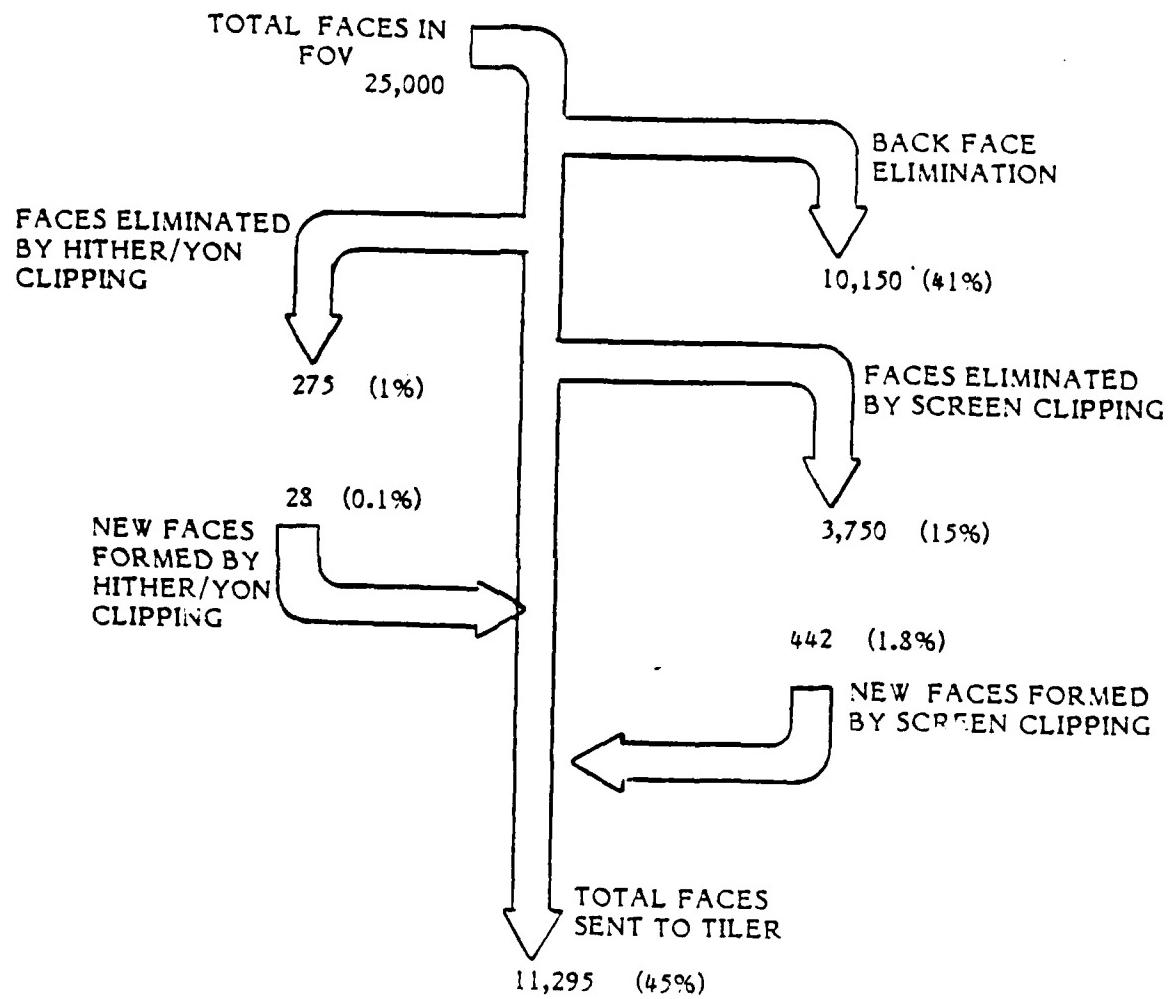


Figure 95. Flow of Faces in Field of View to Tiler
(Typical Frame)

software execution time for a typical image, but play an important role in the hardware implementation.

In addition, it was observed that screen clipping begins to dominate pretiler processing when subimages are pieced together to form a single large image, as illustrated in Figure 96. In this example, the use of sixteen subimages creates sixteen times the number of screen boundaries, and can therefore be expected to require as much as sixteen times the amount of screen clipping. The fractal mountain shown in Figure 66 was generated using this technique. In this case, use of circumscribing-box clipping reduced the image generation time by approximately 10%.

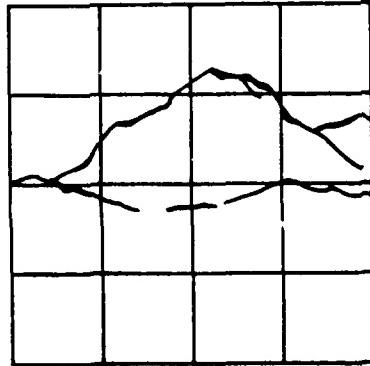
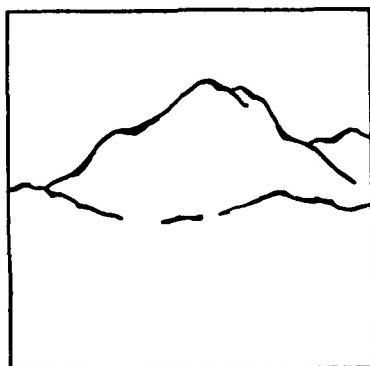


Figure 96. Use of Sixteen Subimages to Form a Single Large Image

The fastest pretiler task is screen projection. The data presented in Table 26 suggest a method to improve the performance even further. Since a high percentage of the vertices in an object are shared among multiple triangles, the projection need not be recomputed for each triangle. Instead, after a projection is initially computed, it can be stored in a buffer and subsequently retrieved. Unfortunately, Table 27 demonstrates that the overhead associated with storing and retrieving the projected coordinates almost entirely offsets the gain through computational savings. Since projection accounts for only a small proportion of the pretiler execution time, use of a vertex buffer was not found to be worthwhile.

Table 26. Screen Space Projection - Shared Vertices
(Typical Frame)

	NUMBER VERTICES	PERCENTAGE
TOTAL	80,000	100.0
COMPUTED	23,200	29.0
SHARED	56,800	71.0

Table 27. Speed Improvement using Vertex Buffer for Projection
(Typical Frame)

	EVALUATION TIME IN MICROSECONDS	PERCENTAGE
WITHOUT VERTEX BUFFER	36	-
WITH VERTEX BUFFER	33	-
SAVINGS	3	8%

In summary, the major bottlenecks in the pretiler appear to be the tiler set-up and the hither/yon trivial test. Further optimization of these tasks would pay immediate dividends in improved system throughput. Precision problems in the tiler set-up, however, complicate its optimization. Use of a vertex buffer to store projected screen values appears to provide no significant improvement in execution time, and should be eliminated. On the other hand, use of the improved screen clipping algorithm can greatly enhance performance, and provides a substantial algorithmic simplification.

7.2.3 PERFORMANCE OF OBJECT PROCESSORS

The terrain and model object processors perform three basic tasks: elimination of back faces, transformation into viewing space, and calculation of surface illumination. Typically, the combination of the two processors consumes less than half the image-rendering time (see Figure 94). Nevertheless, their processing requirements would exceed the capabilities of a general-purpose computer for a real-time CIG system, and they, therefore, also require a hardware implementation.

The most time-consuming task in both object processors is calculating the surface illumination. This process has been substantially simplified by using an inverse transformation to transform the sun vector from world space to model space. In this fashion, the transformation need only be applied once per model, rather than transforming a vertex normal for each vertex. However, additional optimization could substantially reduce execution time of the object processors, especially the terrain object processor.

The back face elimination and viewing transformation tasks are greatly simplified by using a regular grid structure. The terrain object processor exploits this fact, and Figures 93 and 94 and Table 28 demonstrate the results. Exploitation of the grid structure of terrain improves the speed of these two tasks by a factor of five or six. It is clear that implementation of a separate terrain object processor is easily justified.

Table 28. Comparison of Model and Terrain Processing Algorithms
When Processing an Equal Number of Faces

OPERATION	MICROSECONDS		SPEED UP FACTOR
	MOP TIME	TOP TIME	
BACK FACE ELIMINATION	66	11	6.0
VIEWING TRANSFORMATION	70	14	5.0
TOTAL	136	25	5.4

LOD processing was implemented only in the terrain object processor (see Section 6.5). The technique used involved dynamic downsampling of the terrain grid, with overlapping LOD regions to avoid boundary mismatches. The LOD ranges were chosen so that no significant visual degradation was perceived. Table 29 and Figure 97 demonstrate that LOD control can reduce the number of triangles by nearly a factor of four. These savings represent valuable processing power which can be applied toward providing additional scene content.

Table 29. LOD Control by Face Count and Range
(Typical Frame)

	FACES	MAX RANGE (METERS)
TOTAL NO LOD	95,000	-
TOTAL WITH LOD	27,265	-
LOD 1	10,224	10000.0
LOD 2	5,099	14900.0
LOD 3	2,836	19240.0
LOD 4	3,681	28480.0
LOD 5	2,399	50000.0
DOUBLY PROCESSED	3,026	-

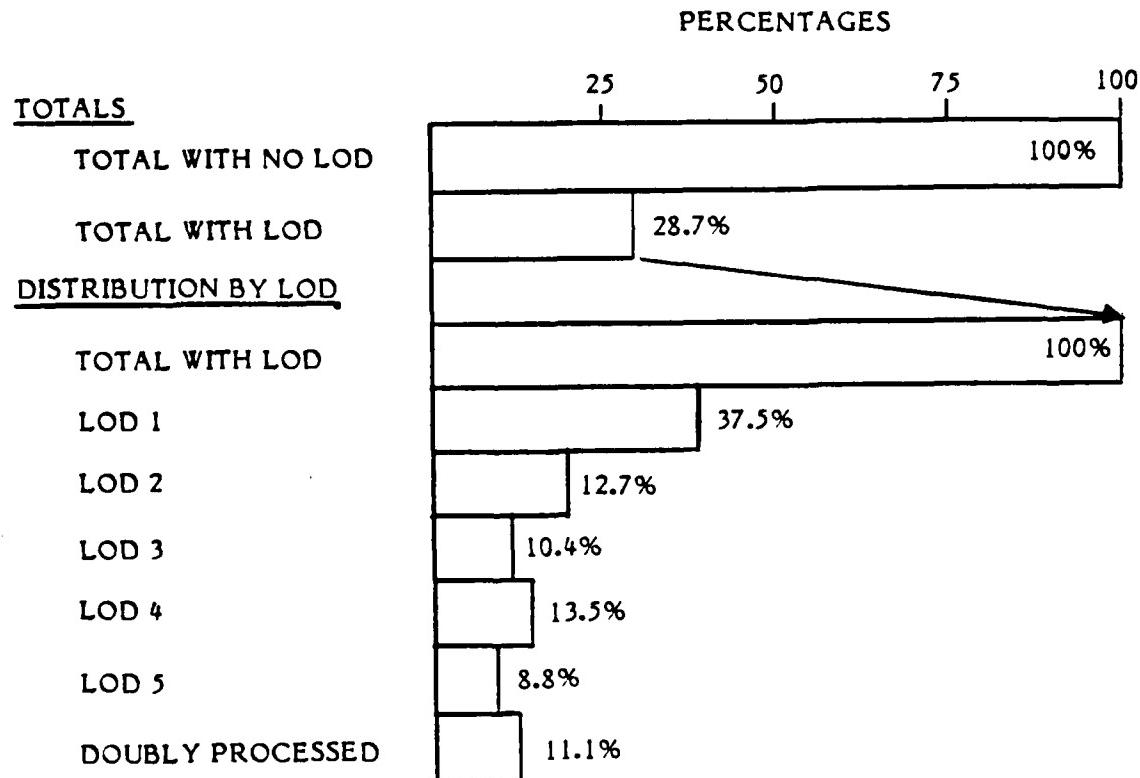


Figure 97. Distribution of Faces in FOV by Level of Detail
(Typical Frame)

The current method of bridging LOD boundaries accounts for approximately 10% of the terrain triangles used. Two methods to avoid this burden were discussed in Section 6.5; however, both methods involved substantial computational overhead in other forms. Another approach is to retain double processing, and reduce the

number of detail levels and transition zones. One could then set the LOD boundaries to achieve a reasonable compromise between triangle counts and visual appeal.

7.2.4 CONCLUSION

Algorithm and code optimization of several processes could be used to substantially improve system performance. Key tasks which deserve attention are the tiler set-up, hither/yon clipping, and surface illumination calculation.

Particularly effective in improving system throughput were LOD control and grid processing. LOD processing should be developed further, especially for models. Although the memory manager and scheduler have only recently been implemented and insufficient data is available to measure their performance, they too should substantially reduce processing time by reducing the amount of disk I/O.

8.0 CONCLUSION

This section summarizes our participation in the second phase of this program into three areas of discussion - problems that were encountered and solutions under consideration, program accomplishments, and future goals.

8.1 PROBLEM AREAS

As with any development effort, problems were encountered in the detailed design phase of this program. Most of these problems were addressed in the sections describing the individual design; however, certain problems have an impact on the entire system. These problems will be identified in this section.

The first major difficulty concerned the antialiasing scheme. The 4:1 oversampling and nine-point pyramid filter provided insufficient filtering to prevent aliasing for small features or dynamic sequences. Consequently, Boeing has sponsored additional research using static pictures, 16 mm film, and video and real-time test sequences. This research indicates that higher oversampling ratios and larger filters can provide excellent antialiasing. Several techniques to prevent system cost escalation in proportion to the oversampling ratio have also been developed. A combination of these techniques should lead to a cost-effective solution to the antialiasing problem.

Another major concern is the number of interconnections in the distribution tree (see Section 4.3). The width of the data word which needs to be routed through the network, combined with the large number of switches which need to be interconnected, results in over one thousand interconnections. In the future, the width of the data word will be widened to accommodate full color, and the number of switches will be increased to accommodate more parallel processors. The interconnection problem is therefore expected to worsen. An acceptable solution to this problem has not yet been found.

The final major issue is how to process transparent objects. Texture maps with transparency could be used to simulate a variety of objects ranging from trees to clouds. But more importantly, transparency would allow 3-D cultural features to fade from one LOD to another by making one LOD progressively more opaque while another LOD becomes progressively more translucent. This technique is employed in Evans and Sutherland's CT-5 system and is considered the most effective LOD transition mechanism in the industry. Neither of the transparency algorithms described in this report are suited to these applications. However, a technique

based on an outgrowth of Boeing's antialiasing work holds great promise. Therefore, it is felt that the transparency issue, and thereby the LOD transition issue, has been resolved.

In summary, although each of these issues does pose a technical difficulty, none of them appear to be insoluble. Thus, the architecture chosen appears to be fundamentally appropriate for CIG processing.

8.2 ACCOMPLISHMENTS

The most tangible accomplishment of this phase of the research program is the design and construction of the special-purpose hardware. The detailed design of the tilers, the distribution tree, and depth/frame buffer memory forced many algorithmic uncertainties to be resolved. This was especially true of the tiling algorithm. The hardware provides tangible evidence of the feasibility of Boeing's CIG architecture. The design experience and performance statistics from this phase of the program can be used to guide the development of the next phase.

Another accomplishment is the design and coding of the software system. Many alternative designs were considered, and the design chosen was enhanced by new algorithms or novel adaptations of previously known algorithms. The current design of the software system will largely determine the design of future hardware.

A third accomplishment is the investigation of other topics in computer graphics and how they relate to the depth-buffer architecture. In some cases, such as texture mapping, these investigations have led to an algorithm which is ready for hardware implementation. In other cases, such as curved surfaces, this investigation has constructed a technology foundation upon which future, more directed efforts can build.

The final major accomplishment is the design of a database structure which supports efficient CIG processing. In this role, it is an integral component of the CIG architecture. The flexibility of its design will allow the database structure to accommodate future developments and applications.

In summary, the major accomplishment of this phase of the program is not simply the completion of a research task. It is also the anticipation of future applications and developments of this new CIG architecture.

8.3 FUTURE DEVELOPMENT

This phase of the development effort has brought the Boeing CIG architecture from concept to prototype. The next step is to apply the concepts developed and experience gained, and to develop a system for use outside the laboratory. The exact application chosen for this system will largely influence many of the detailed design choices. However, certain key features of the architecture will remain constant.

The application chosen will determine, among others, the processing speed of the system, the type of processors needed, and the nature of the ancillary software. The processing speed needed for a flight simulation system, which must produce approximately thirty frames per second, is vastly different from the speed needed for a command and control system, which may only need to produce a frame every three seconds. The required processing speed may also affect the choice of processors. For example, for a flight simulator a database composed of triangles, which are easy to process, is entirely satisfactory. For a CAD/CAM application, however, curved surface rendering is essential. The application directly affects the ancillary software, such as the database construction tools, since these form the direct interface between the application and the CIG system.

Other features of the CIG system will remain constant. Clearly, an intelligent memory to implement the z-buffer algorithm will form the basis of any CIG system based upon this architecture. The use of parallel processing, and hence the use of a network to resolve contention, is another key feature. Finally, a standardized database structure will be maintained, thus permitting maximum commonality of database construction software.

Standardization of these key issues will lead to a family of compatible CIG systems. This family will make optimal use of parallel processing, be sufficiently flexible to accommodate a variety of object data types, and will be scalable to suit the needs of a wide variety of applications.

9.0 REFERENCES

- Barn83 Barnes, G. H., and Lundstron, S. F. 1983. Design and validation of a connection network for many-processor multiprocessor system. IEEE Computer, December, p. 31.
- Barr81 Barr, Alan H. 1981. Superquadrics and angle-preserving transformations. Computer Graphics and Applications. 1 (January):11-23.
- Bars83 Barsky, Brian, and Beatty, John. 1983. Local control of bias and tension in beta-splines. ACM Transactions on Graphics. 2 (April):109-134.
- Bask76 Baskett, F., and Smith, A. J. 1976. Interference in multiprocessor computer systems. Comm. ACM. 19 (June):327.
- Batc76 Batcher, K. E. 1976. The flip network in STARAN. Proc. 1976 Int'l Conf. Parallel Processing, August, p. 65.
- Berr80 Berry, M. V., and Lewis, Z. V. 1980. On the Weierstrass-Mandelbrot fractal function. Proc. Royal Society of London. A 370:459-484.
- Besi37 Besicovitch, A. S., and Ursel, H. D. 1937. Sets of fractional dimensions (v): On dimensional numbers of some continuous curves. Journal of the London Mathematical Society. 12:18-25.
- Bhan75 Bhandarkar, D. P. 1975. Analysis of memory interference in multiprocessors. IEEE Trans. on Computers. C-24:897.
- Blin77a Blinn, James. 1977a. Models of light reflection for computer synthesized pictures. Computer Graphics, SIGGRAPH 1977 Proc. 11:192-198.
- Blin77b _____ . 1977b. A homogeneous formulation for lines in 3 space. Computer Graphics, SIGGRAPH 1977 Proc. 11:237-241.

- Blin78a _____. 1978a. Simulation of wrinkled surfaces. Computer Graphics, SIGGRAPH 1978 Proc. 12 (August):286-292.
- Blin78b Blinn, J. F., and Newell, M. E. 1978b. Clipping using homogeneous coordinates. Computer Graphics, SIGGRAPH 1978 Proc. 12 (August):245-251.
- Blin82 Blinn, J. F. 1982. Light reflection functions for simulation of clouds and dusty surfaces. Computer Graphics, SIGGRAPH 1982 Proc. 16 (July):21-29.
- Boo83 Booth, Kellogg S., Kochanek, Doris H., and Wein, Marceli. 1983. Computer animated films and video. IEEE Spectrum. 20 (February):44-51.
- Bouk69 Bouknight, W. J. 1969. An improved procedure for generation of half-tone computer graphics representations. University of Illinois, Coordinated Science Laboratory. R-432, September.
- Bouk70 _____. 1970. A procedure for generation of three-dimensional half-toned computer graphics representations. Comm. ACM. 13 (September):527.
- Bui75a Bui, Tuong-Phong. 1975a. Illumination for computer-generated images. Comm. ACM. 18 (June):311-317.
- Bui75b Bui, Tuong-Phong, and Crow, Franklin C. 1975b. Improved rendition of polygonal models of curved surfaces. Proc. Second USA-JAPAN Computer Conf. August 1975, pp. 475-480.
- Bunk82 Bunker, W. Marvin. 1982. Training in a simulated world. Computer Graphics World. 5 (December):35-42.
- Carp80 Carpenter, Loren. 1980. Vol libre. Computer-generated animated movie. First showing at SIGGRAPH 1980, Seattle, Washington (July).

- Catm78 Catmull, Edwin E. 1978. A hidden-surface algorithm with anti-aliasing. Computer Graphics, SIGGRAPH 1978 Proc. 12 (August):6-11.
- Chan77 Chang, D. Y., Kuck, D. J., and Lawrie, D. H. 1977. On the effective bandwidth of parallel memories. IEEE Trans. on Computers, C-26:480.
- Clar76 Clark, James H. 1976. Hierarchical geometric models for visible surface algorithms. Comm. ACM. 19 (October):547-554.
- Cohe80 Cohen, Elaine, Lyche, Tom, and Riesenfeld, Richard. 1980. Discrete b-splines and subdivision techniques in computer-aided geometric design and computer graphics. Computer Graphics and Image Processing.
- Crow77 Crow, Franklin C. 1977. The aliasing problem in computer-generated shaded images. Comm. ACM. 20 (November):799-805.
- Crow81 _____ 1981. A comparison of anti-aliasing techniques. Computer Graphics and Applications. 1 (January):40-48.
- Crow82 _____ 1982. Computational issues for rendering anti-aliased detail. Proc. IEEE Spring Comp. Conf., February.
- Cyru78 Cyrus, M. L., and Beck, J. 1978. Generalized two- and three-dimensional clipping. Computer and Graphics. 3:23-28.
- Debo78 DeBoor, Carl. 1978. A practical guide to splines. New York: Springer-Verlag.
- Dias81 Dias, D. M., and Jump, J. R. 1981. Analysis and simulation of buffered delta networks. IEEE Trans. on Computers. C-30:273.

- Duff79 Duff, Tom. 1979. Smoothly shaded renderings of polyhedral objects on raster displays. Computer Grahpics, SIGGRAPH 1979 Proc. 13:270-275.
- Feib80 Feibusch, Eliot A. et al. 1980. Synthetic texturing using digital filters. Computer Graphics, SIGGRAPH Proc. 1980. 14 (July):294-301.
- Feng73 Feng, T. 1973. Parallel processing characteristics and implementation of data manipulating functions. Rome Air Development Centre Report, RADC-TR-73-189, July.
- Fole82 Foley, J. D., and Van Dam, A. 1982. Fundamentals of interactive computer graphics. Reading, Massachusetts: Addison-Wesley.
- Four80 Fournier, Alain. 1980. Stochastic modeling in computer graphics. Ph.D. Dissertation, University of Texas at Dallas.
- Four82 Fournier, Alain, Fussell, Don, and Carpenter, Loren. 1982. Computer rendering of stochastic models. Comm. ACM. 25 (June):371-384.
- Gene68 General Electric Corporation. 1968. Modifications to interim visual spaceflight simulator. Final Report, NASA Contract NAS 9-3916, February.
- Gene71 General Electric Corporation. 1971. Electronic scene generator expansion system. Final Report, NASA Contract NAS 9-11065, December.
- Gord74 Gordon, William, and Riesenfeld, Richard. 1974. B-spline curves and surfaces. Computer Aided Geometric Design. New York: Academic Press.
- Gott83 Gottlieb, A. G. et al. 1983. The NYU ultracomputer - designing an MIMD shared memory parallel computer. IEEE Trans. on Computers. C-32:175.

AD-A141 883

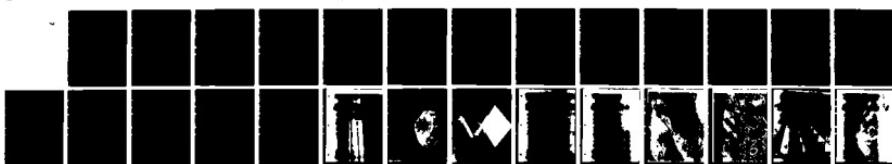
MULTIPROCESSOR Z-BUFFER ARCHITECTURE FOR HIGH-SPEED
HIGH COMPLEXITY COMPUTER IMAGE GENERATION(U) BOEING
AEROSPACE CO SEATTLE WA DEC 83 MDA903-82-C-0101

F/G 9/2

4-4

UNCLASSIFIED

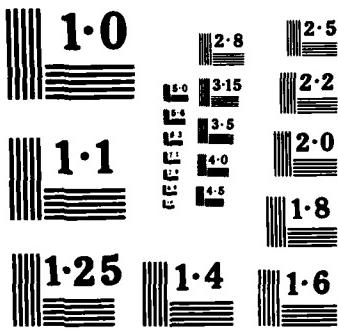
NL



END

FILMED

0910



- Gour71 Gouraud, H. 1971. Computer display of curved surfaces. University of Utah, Department of Computer Science, UEC-CSC-71-113, June.
- Gupt81 Gupta, Satish, and Sproull, Robert. 1981. Filtering edges for grey-scale displays. Computer Graphics, SIGGRAPH 1981 Proc. 15 (August):1-5.
- Horo82 Horowitz, E., and Shani, S. 1982. Fundamentals of data structures, pp. 114-117. Rockville, Md.: Computer Science Press, Inc.
- Hutc81 Hutchinson, John E. 1981. Fractals and self similarity. Indiana University Mathematics Journal, 30:713-747.
- Jaec82 Jaech, Jeremy. 1982. Rendering solid-shaded surfaces represented as rational b-splines. Boeing Computer Services. Unpublished.
- Keit81 Keith, Stephan R. 1981. A transformation structure for animated 3-D computer graphics. Computer Graphics, SIGGRAPH 1981 Proc. 15 (April):72-91.
- Keji83 Kejiya, James T. 1983. New techniques for ray tracing procedurally defined objects. Computer Graphics, SIGGRAPH 1983 Proc. 17 (July):91-102.
- Klei76 Kleinrock, L. 1976. Queueing systems, vol. 2. New York: John Wiley & Sons.
- Lane80 Lane, Jeffery and Riesenfeld, Richard. 1980. A theoretical development for the computer generation and display of piecewise polynomial surfaces. IEEE Transactions on Pattern Analysis and Machine Intelligence, January.
- Lang82 Lang, T., Valero, M., and Alegre, I. 1982. Bandwidth crossbar and multiple-bus connections for multiprocessors. IEEE Trans. on Computers. C-31:1227.

- Lawr75 Lawrie, D. H. 1975. Access and alignment of data in an array processor. IEEE Trans. on Computers. C-24:1145.
- Mach78 Machover, C. 1978. A brief, personal history of computer graphics. Computer, 11 (November).
- Maek81 Maekawa, M. 1981. Optimal processor interconnection topologies. IEEE Symposium on Computer Architecture, p. 171.
- Mand68 Mandelbrot, B. B., and Van Ness, J. W. 1968. Fractional Brownian motions, fractional noises and applications. SIAM Review. 10 (October):422-437.
- Mand71 Mandelbrot, B. B. 1971. A fast fractional Gaussian noise generation. Water Resources Research 7, pp. 543-553.
- Mand75 _____. 1975. Stochastic models for the earth's relief, the shape and fractal dimension of coastlines, and the number area rule for islands. Proc. Nat. Acad. Sci. USA. 72 (October):2825-2828.
- Mand77 _____. 1977. Fractals: form, change and dimension. San Francisco: Freeman.
- Mand82a _____. 1982a. The fractal geometry of nature. San Francisco: Freeman.
- Mand82b Mandelbrot, B. B., Fournier, A., Fussell, Don, and Carpenter, Loren. 1982b. Technical correspondence, 'comment on computer rendering of fractal stochastic models'. Comm. ACM. 25 (August):581-584.
- Mars83a Marsan, M. A. et al. 1983a. Modeling bus contention and memory interference in a multiprocessor system. IEEE Trans. on Computers. C-32:61.

- Mars83b Marsan, M. A., Balbo, G., and Conte, G. 1983b. Comparative performance analysis of single bus multiprocessor architectures. IEEE Trans. on Computers. C-31:1179.
- Matt80 Matthis, Larry. 1980. Parametric subdivision: a brief survey on an implementation. University of Waterloo, December.
- Mira82 Mirante, Anthony, and Weingarten, Nicholas. 1982. The radical sweep algorithm for constructing triangulated irregular networks. Computer Graphics and Applications. 2 (May):11-21.
- Mokh82 Mokhoff, Nicolas. Superminis give mainframes a run for their money. Digital Design, October. Reprint.
- Newe72 Newell, M. E., Newell, R. G., Sancha, T. L. 1972. A new approach to the shaded picture problem. Proc. ACM National Conf.
- Newe77 Newell, Martin E., and Blinn, James F. 1977. The progression of realism in computer generated images. Proc. ACM Annual Conf., pp. 444-448.
- Newm79 Newman, William M., and Sproull, Robert F. 1979. Principles of interactive computer graphics, 2nd ed., New York: McGraw-Hill.
- Nort82 Norton, Alan. 1982. Generation and display of geometric fractals in 3-D. Computer Graphics, SIGGRAPH 1982 Proc. 16 (July):61-68.
- ParkF80 Parke, F. 1980. Simulation and expected performance analysis of multiple processor z-buffer systems. Computer Graphics, SIGGRAPH 1980 Proc. 14 (July):48-56.
- ParkD80 Parker, D. S. 1980. Notes on shuffle/exchange-type switching networks. IEEE Trans. on Computers. C-29:213.

- Pate81 Patel, J. H. 1981. Performance of processor-memory interconnection for multiprocessors. IEEE Trans. on Computers. C-30:771.
- Peas77 Pease, M. C. 1977. The indirect binary n-cube microprocessor array. IEEE Trans. on Computers. C-26:548.
- Prem80 Premkumar, U. V. et al. 1980. Design and implementation of the Banyan interconnection network in TRAC. 1980 National Computer Conf., p. 643.
- Reed83 Reed, D. A., and Schwetman, H. D. 1983. Cost performance bounds for multimicrocomputer networks. IEEE Trans. on Computers. C-32:83.
- Reev83 Reeves, William T. 1983. Particle systems - a technique for modeling a class of fuzzy objects. Computer Graphics, SIGGRAPH 1983 Proc. 17 (July):359-376.
- Reis83 Reisman, Arnold. 1983. Device, circuit, and technology scaling to micron and submicron dimensions. Proc. IEEE Conf. 71 (May):550-565.
- Romn70 Romney, G. W. 1970. Computer assisted assembly and rendering of solids. University of Utah, Department of Computer Science. TR-4-20.
- Rubi72 Rubin, Steven M. 1972. The representation and display of scenes with a wide range of detail. Computer Graphics and Image Processing. 19 (July):291-298.
- Rubi80 Rubin, Steven M. and Whitted, Turner. 1980. A 3-dimensional representation for fast rendering of complex scenes. Computer Graphics, SIGGRAPH 1980 Proc. 14 (July):110-116.

- Scha81 Schachter, Bruce J. 1981. Computer image generation for flight simulation. Computer Graphics and Applications, October, pp. 29-68.
- Scha83 _____ . 1983. Computer Image Generation. New York: John Wiley & Sons.
- Schu69 Schumacker, R. A. et al. 1969. Study for applying computer-generated images to visual simulation. AFHRL TR-69-14. U.S. Air Force Human Resources Laboratory, September.
- Sieg80 Siegel, H. J. '80. The theory underlying the partitioning of permutation networks. IEEE Trans. on Computers. C-29:791.
- Spro68 Sproull, R. F., and Sutherland, I. E. 1968. A clipping divider. Proc. AFIPS FJCC 1968 Conf. 33:765-775.
- Stei83 Stein, Kenneth J. 1983. Computer image system advances. Aviation Week and Space Technology, September 27, pp. 73-75.
- Suth74a Sutherland, I. E., and Hodgman, G. W. 1974a. Reentrant polygon clipping. Comm. ACM. 17 (January):32-42.
- Suth74b Sutherland, I. E., Sproull, R., and Schumacker, R. A. 1974b. A characterization of ten hidden-surface algorithms. ACM Computing Surveys, March.
- Warn69 Warnock, J. E. 1969. A hidden-surface algorithm for computer-generated halftone pictures. University of Utah, Department of Computer Science. TR-4-15, June.
- Watk70 Watkins, G. S. 1970. A real-time visible surface algorithm. University of Utah, Department of Computer Science. UTECH-CSC-70-101, June.

- Weil77 Weiler, K., and Atheron, P. 1977. Hidden-surface removal using polygon area sorting. Computer Graphics, SIGGRAPH 1977 Proc. 11:214.
- Whit82 Whitted, Turner. 1982. Some recent advances in computer graphics. Science. 215 (February):767-774.
- Wild71 Wild, C., Rougelot, R. S., and Schumacker, R. A. 1971. Computing full color perspective images. General Electric Technical Information Series R7IELS-26, May.
- Will83 Williams, Lance. 1983. Pyramidal parametrics. Computer Graphics, SIGGRAPH 1983 Proc. 17(July):1-11.
- Wu80a Wu, C. L., and Feng, T. Y. 1980a. On a class of multistage interconnection networks. IEEE Trans. on Computers. C-29:694.
- Wu80b _____. 1980b. The reverse-exchange interconnection networks. IEEE Trans. on Computers. C-29:801.
- Wu80c _____. 1980c. On a distributed-processor communication architecture. Proc. Compson, Fall, p. 599.
- Wylie67 Wylie, C. et al. 1967. Halftone perspective drawings by computer. Proc. AFIPS FJCC 1967 Conf. 31:49.

APPENDIX A - HISTORICAL BACKGROUND

Computer image generation (CIG) technology is a dynamic discipline that has been the focus of intense research and development, and, as a result, has experienced phenomenal growth. This section presents a brief overview of the birth and evolution of CIG. It begins with a history of the broader field of computer graphics that stretches from its rudimentary beginnings in the Whirlwind I computer to the sophisticated, high-complexity techniques of CIG today. An examination of the influence of flight simulation on the course of CIG along with an analysis of the technological trends that shape its future is also included here.

A.1 GRAPHICS RESEARCH

Computer graphics can trace its beginnings back to 1950 when the Whirlwind I computer became operational. This facility plotted points onto a cathode-ray tube (CRT) and automatically photographed the resulting graph. Now, 33 years later, this same fundamental process is still being performed.

A milestone in the development of interactive computer graphics was Ivan Sutherland's 1963 MIT doctoral thesis describing SKETCHPAD. This system was the first to exhibit data structures that permitted efficient interactive construction and manipulation of objects.

In the late 1960's, researchers began to develop algorithms to generate shaded images of solid objects. Gouraud introduced a simple but effective way to smoothly shade polygonal objects. This work was followed by Phong who introduced the concept of interpolating surface normals in order to create highlights for a more realistic appearance.

In the early 1970's, Catmull developed an algorithm that rendered curved surfaces defined with bicubic Bezier patches. This algorithm employed a recursive subdivision technique that divided the patch until it could be represented by a single pixel. He resolved the problem of hidden surfaces with use of the "z-buffer" algorithm.

In 1974, Sutherland, Sproull, and Schumacker compiled ten hidden-surface algorithms for a paper that appeared in Computing Surveys (Suth74b). Several alternative solutions to the hidden-surface problem were definitely available by that point in time.

The work of Blinn has been a milestone for realism. His lighting models are the foundations for all realistic graphics today. He was also the first to introduce the concept of mapping texture in the form of surface normals for the appearance of wrinkled surfaces (Blin78a).

Toward the end of the 1970's, the topic of antialiasing was researched intensely. Considerable literature can be found in the recent SIGGRAPH proceedings on the

topic of antialiasing in time and space dimensions. The topic of shadows also received considerable attention during this time period.

Recent research topics have included: the simulation of light by "ray tracing"; simulation of natural phenomenon by fractals and particle systems; machine architectures based on various forms of concurrency and VLSI structures; and the modeling of the human body.

For a more detailed history of computer graphics see (Boot83), (Bunk82), (Fole82), (Mach78), (Newe77), (Newm79), and (Whit82).

topic of antialiasing in time and space dimensions. The topic of shadows also received considerable attention during this time period.

Recent research topics have included: the simulation of light by "ray tracing"; simulation of natural phenomenon by fractals and particle systems; machine architectures based on various forms of concurrency and VLSI structures; and the modeling of the human body.

For a more detailed history of computer graphics see (Boot83), (Bunk82), (Fole82), (Mach78), (Newe77), (Newm79), and (Whit82).

A.2 FLIGHT SIMULATION

Flight simulators are devices in which flight crews can receive training without actually flying. They are not intended to be a realistic substitute for flying an aircraft. They are training devices meant to bridge the gap between classroom instruction and real-world operations. They are also one of the most important applications of computer image generation (CIG) technology today. The following discussion is developed in (Scha81) and (Scha83).

In their earliest and crudest forms, flight simulators were primarily instrument trainers with no visual systems and a minimal motion system. In their most sophisticated form today, flight simulators emulate the instrumentation of the cockpit, motion and sound, gravitational forces, radar and sensor displays, and visuals of real flight. These visuals can include the simulation of textures, shadows, clouds, haze, weather, night conditions, and light paths. This evolution in visual systems is the story of CIG.

The flight simulators based on CIG that have emerged in the last decade have centered around a basic architecture consisting of three pipelined stages. The first stage is usually a general-purpose 32-bit computer such as a VAX, SEL, or Perkin-Elmer. This computer usually receives position and attitude information from the flight computer and uses this information to retrieve data blocks from a visual three-dimensional database. The second stage is typically a special-purpose computer that employs parallelism. This computer performs image generation tasks which include perspective projection, hidden-surface elimination, transformation, clipping, and screen assignment. The third stage is the video processor which converts a two-dimensional digital image into analog video.

The first device using these CIG techniques was produced by General Electric in the late 1950's. It combined a calligraphic display with analog circuitry to produce a pattern on a flat ground plane. It was essentially a pioneering effort to prove the feasibility of real-time graphics.

In 1962, GE was funded by NASA to build a CIG flight simulator suitable for training. Again, GE used a flat ground plane as the basic surface representation. Polygonal ground features and polyhedral objects were placed on this surface. No

edge smoothing was employed to blend the polygonal surfaces, and no attempt was made to combine single scenes into a wider field of view. The results of this simulation were little more than a checkered earth surface, but it served as a starting point for the development of the capability to display simple objects.

The 1970's saw an explosion in CIG technology as applied to the problems of flight simulation. Four companies - GE, Singer/Link, Evans and Sutherland, and McDonnell Douglas - basically dominated the field during this decade, and advanced the technology dramatically.

In 1972, GE delivered its Advanced Development Model (ADM) visual flight simulator to the Navy. It was intended to be used as an instrument to measure the training effectiveness of computer-generated imagery. The displayed screen image was a fairly sparse 500-edge scene. The 500-edge limitation and the lack of adequate texturing was found to provide inadequate visual cues for lineups and landings of aircraft. It did accomplish the first simulation of haze.

GE's Advanced Simulator for Undergraduate Pilot Training (ASUPT) was delivered to the Air Force in 1974 for the express purpose of exploring the role of flight simulators in pilot training. It displayed 2500 edges that were smoothed in the horizontal direction. It was the first simulator to use wraparound infinity optics with no breaks between channels, providing for an extraordinarily wide field of view. The resolution of ASUPT's display device was considered marginal, hindered also by the fact that it was monochromatic. ASUPT was typical of simulators of this era in that its rudimentary level-of-detail management and limited edge-display capability caused objects to suddenly pop in or out of view.

GE delivered the first day/night full-color visual flight simulator for commercial aircraft to Boeing in 1975. It had the capability of displaying 1000 edges and 2000 light points and was later upgraded to 4000 edges and 2000 light points. Its field of view ($30^\circ \times 40^\circ$) proved too narrow for maneuvers other than straight-in approaches to runways. It did make use of special effects such as cloud cover, scud clouds, and atmospheric attenuation.

In 1978, GE delivered the Aviation Wide-Angle Visual System (AWAVS) to the Navy for training aircraft carrier pilots. It could display 1000 edges and 2000 light

points as a monochrome, wide-angle, real image produced on the inside of a 10-foot radius dome. This approach had difficulties associated with it since complicated transforms were required to compensate for the dome's distortion.

The latest GE development is their B-52 and C-130 simulators delivered in the early 1980's. They were the first simulators delivered with a 250,000-square-mile database constructed by GE's new automated database development system. They also employed edge-smoothing in both vertical and horizontal directions.

Singer/Link introduced in the mid-1970's the Night Visual System (NVS), a low-cost simulator attachment for generating night scenes. NVS can display 2000 to 6000 light points and can simulate horizon glow, runway tire markings, and atmospheric attenuation. These devices are known for their excellent three-dimensional realism.

Singer/Link has also introduced two versions of the Digital Image Generation (DIG) system. The first version displays 8000 edges, and has been delivered in a variety of configurations to NASA, the Air Force, Northrup Aircraft, All Nippon Airways, and the Army. It offers vertical and horizontal edge smoothing, full color, double-buffer memory, and a landing-light option that brightens scene components within landing-light or taxi-light spheres. The second version was built especially for the B-52 aircraft. It can display up to 12,000 edges and can handle illumination calculations that involve infrared emissions.

Evans and Sutherland have developed two main simulator products - NOVOVIEW and their continuous-tone systems. NOVOVIEW is a low-cost, real-time, night-only system that can generate perspective scenes constructed of 2000 light points and a shaded horizon band. NOVOVIEW has been enhanced several times to increase light point capacity and to add special effects such as stars, moon, an aura surrounding the airport, and glowing sunrise/sunset. Weather effects and the simulation of landing-light illumination were also added to later versions. NOVOVIEW SP1 was introduced in 1977. It could display up to 200 surfaces as well as providing solid and moving objects, directional lights, and a dusk mode capability. NOVOVIEW SP2 is an SP1 with a shadow-mask CRT.

The continuous-tone devices introduced by Evans and Sutherland in 1973 are higher-cost CIG devices that concentrate on scene quality. The first unit (CT-1) was delivered to Case Western Reserve University. It used a high-precision CRT camera station to produce color movies in non-real-time by sequencing color filters. CT-1 could display 400 polygons using Gouraud continuous shading and edge smoothing. The latest version (CT-5A) can display up to 3750 polygons using a feature-sequential approach to image generation rather than a scan-line approach.

McDonnell Douglas's contribution to the CIG market is known as VITAL for Virtual Image Takeoff and Landing. VITAL II was the first unit introduced in 1971, and could display 1200 light points on a black background. VITAL III is a high-resolution upgrade of VITAL II, providing runway surface markings for a more effective simulation. McDonnell Douglas has developed helmet-mounted displays of VITAL IV with Honeywell optics. The current McDonnell Douglas units are known as VITAL VI. This device works with a database paged from disk that consists of light strings and surfaces, including a fully marked runway.

The visuals of flight simulation have been the subject of intense research and development efforts, and monumental progress has been made in the last decade. The next decade should bring this technology to maturity, refining the process and results. In hardware, efforts will be concentrated on improving cost/ performance and speed. Some people in the industry believe that VLSI technology will determine the future of CIG while others are proponents of a more traditional architecture using gate arrays. In software, more economical ways of performing such tasks as antialiasing, hidden-surface removal, field-of-view processing, and texturing will have to be found. The database construction process will have to be standardized and automated using Defense Mapping Agency terrain and culture files for input. Databases will be growing larger as simulators fly further, and current methods of hand-modeling will be inadequate.

A.3 TECHNOLOGY TRENDS

In the next few years, CIG systems will be asked to adapt their architectures to great advances in state-of-the-art technologies. The CIG system will especially be affected by advances in semiconductor and video display technologies.

Semiconductor devices will become faster, more complex, and cheaper as minimum feature sizes are reduced, new processes are refined, and new packaging methods are incorporated. Figure 98 shows how device complexity has nearly doubled every year for the last two decades (Reis83). This rate of improvement is expected to continue for the next decade as the industry is spurred on by economic competition and government programs such as VHSIC (Very High Speed Integrated Circuit).

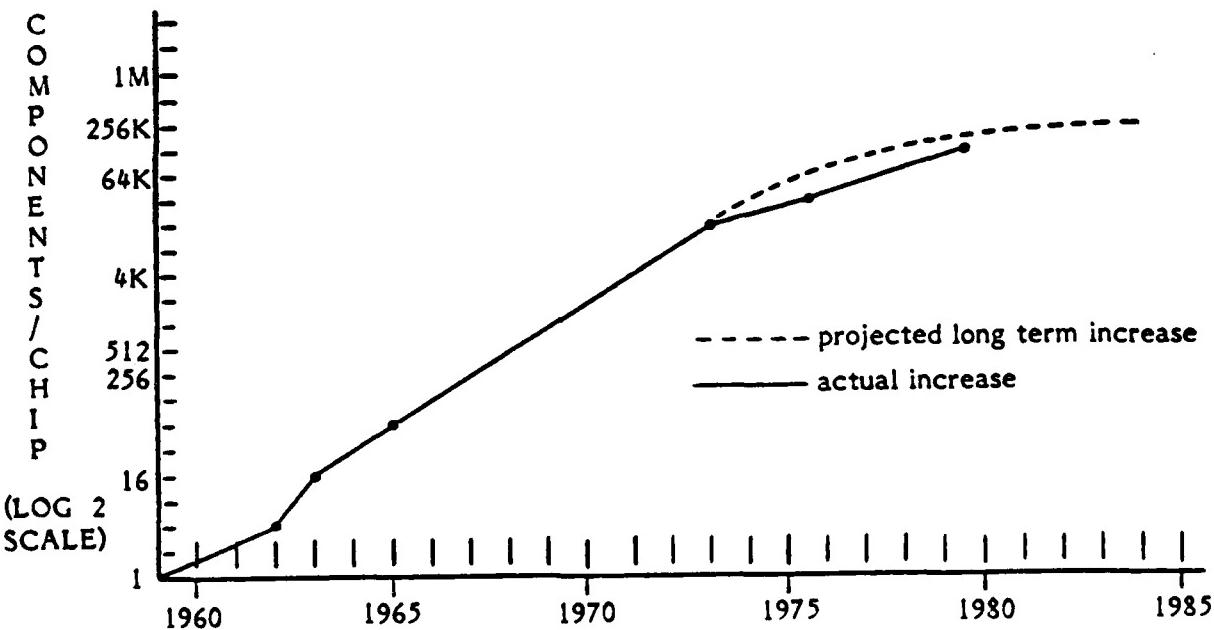


Figure 98. Increases in Device Complexity through Time

Most of the improvements have been made possible by refinements in lithographic techniques that have permitted smaller and smaller minimum feature size. Present state-of-the-art chips use $2-2.5\lambda$ technologies. Soon $1-1.25\lambda$ technology chips will be commercially available, and sub 1λ technology chips have already been demonstrated in the laboratory.

These decreases in minimum feature size will more than double the number of components that can be put on one chip. The increase in components will result in an increase in the number of on-chip operations and memory devices available. This will allow microprocessors to implement wider data paths and provide enhanced capabilities. Increases in memory size will especially benefit the CIG system which requires extensive data storage in both the depth buffer and the memory management section.

For field-effect transistor (FET) technologies such as NMOS and CMOS, smaller feature sizes also mean faster operational speeds. Device speed will also be increased by the use of gallium arsenide (GaAs) instead of silicon as the substrate for the devices. These increases in speed would allow the addition of features that are currently too computationally intensive for incorporation into the system.

For certain applications, such as mathematical functions, where speed is very important and complexity is not great, recently developed chips that use bipolar technologies like ECL can be utilized. Bipolar technologies are very fast, but also consume large amounts of power. This power consumption limits the density of components on a chip, and therefore, the extent to which they can be used.

As integrated circuits become faster, system performance begins to be limited by the input/output (I/O) speed of the chip. This I/O speed will soon be limited by the chip-to-chip interconnection length. Therefore, further increases in data I/O rates will require more or wider data paths resulting in more pins per chip. Recently developed pin grid arrays help solve this problem. They provide up to 144 pins per package, densely packed. This not only increases the number of pins per package, it also decreases the size of the chip.

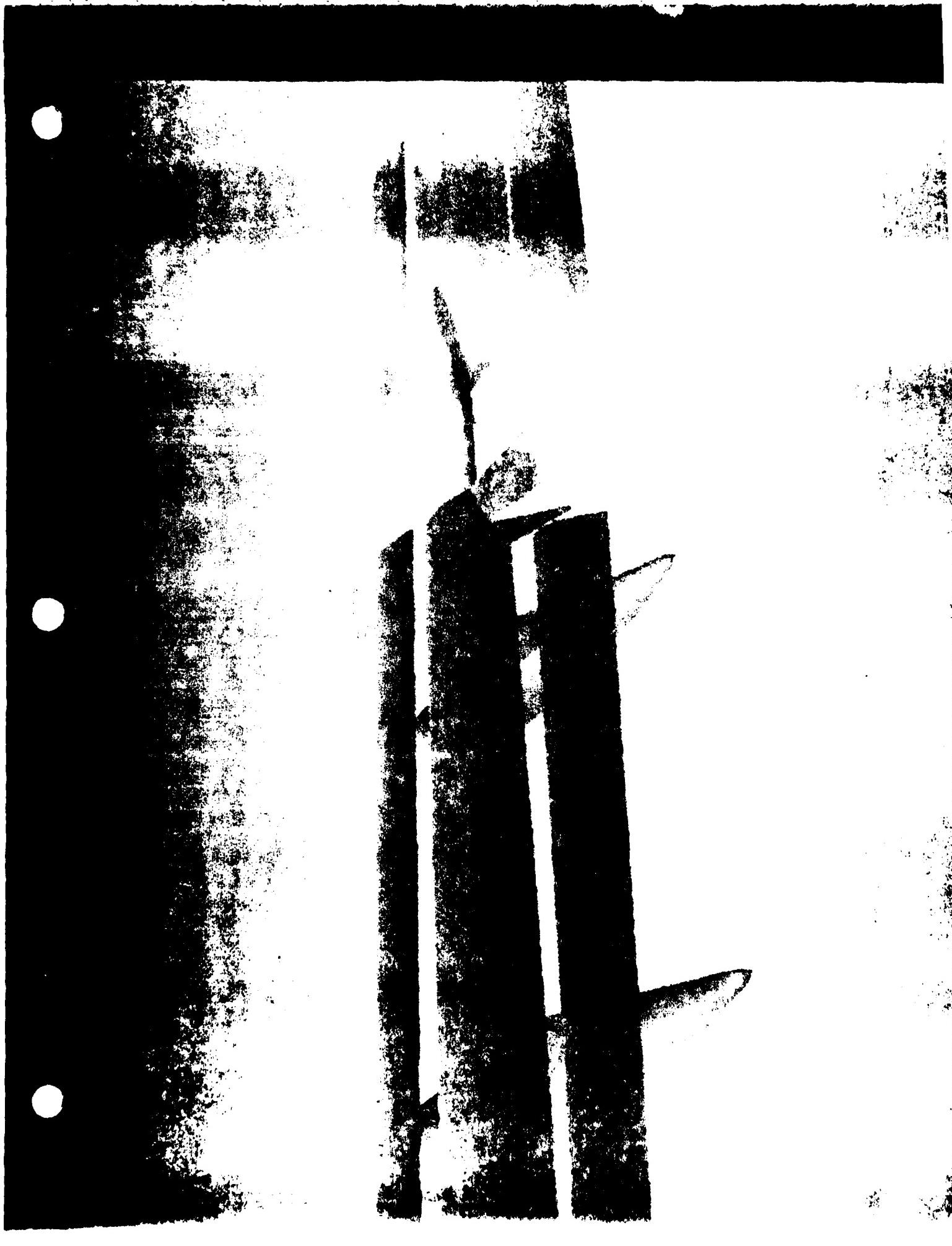
Advances in semiconductor technologies will affect the internal structure of the CIG system. In contrast, advances in video display technologies will affect how the CIG system interfaces with external equipment. Since the CIG system constructs a full frame before any data is displayed, the data can be generated in any order that the video scan scheme requires. Also, the CIG system can be expanded to accommodate different screen sizes. A key feature of the system architecture is that it is designed to accommodate and utilize these advanced technologies as they develop.

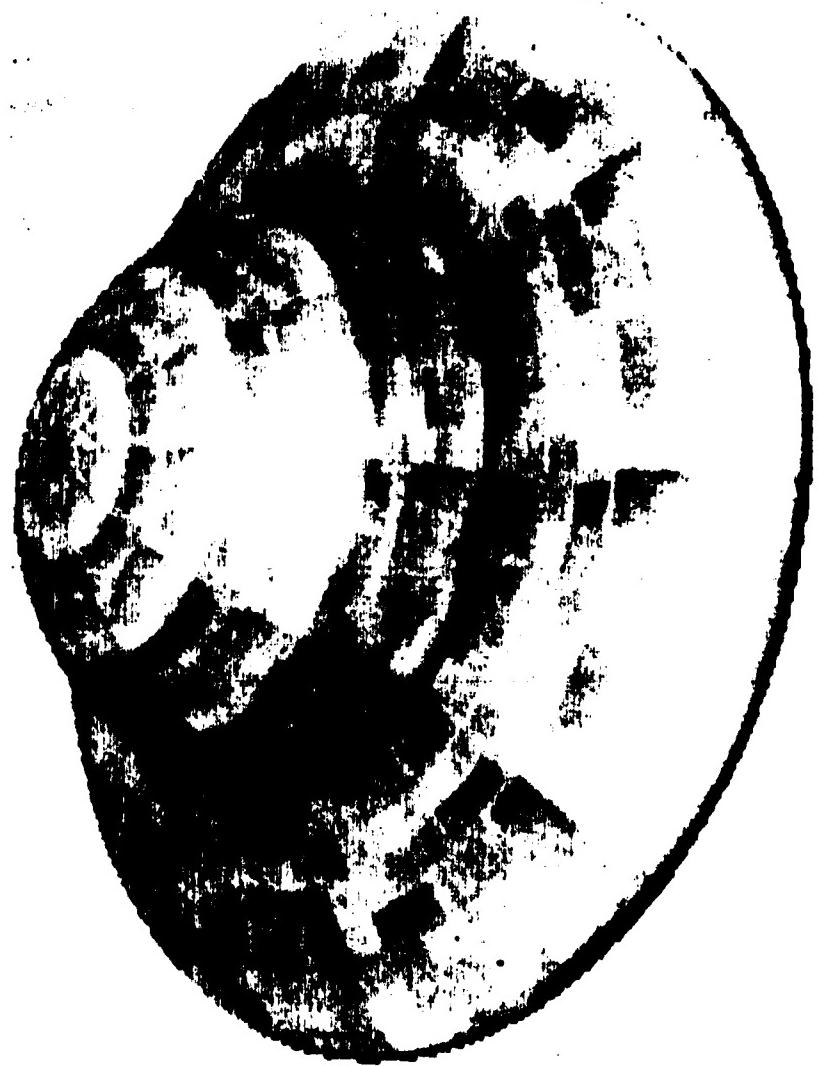
PLATES

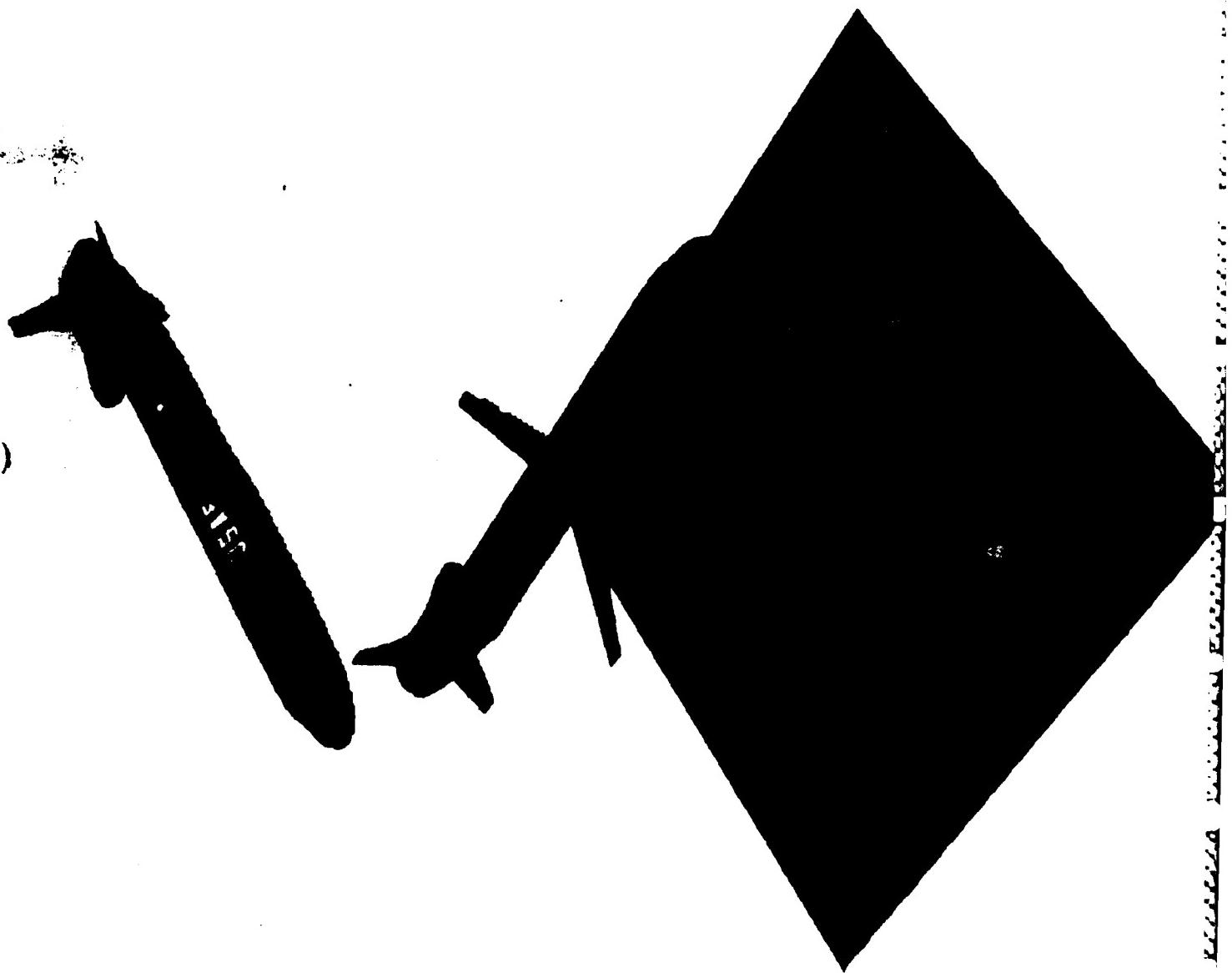
These plates represent several examples of the various types of pictures that have been generated by Advanced Computer Graphics Technology.

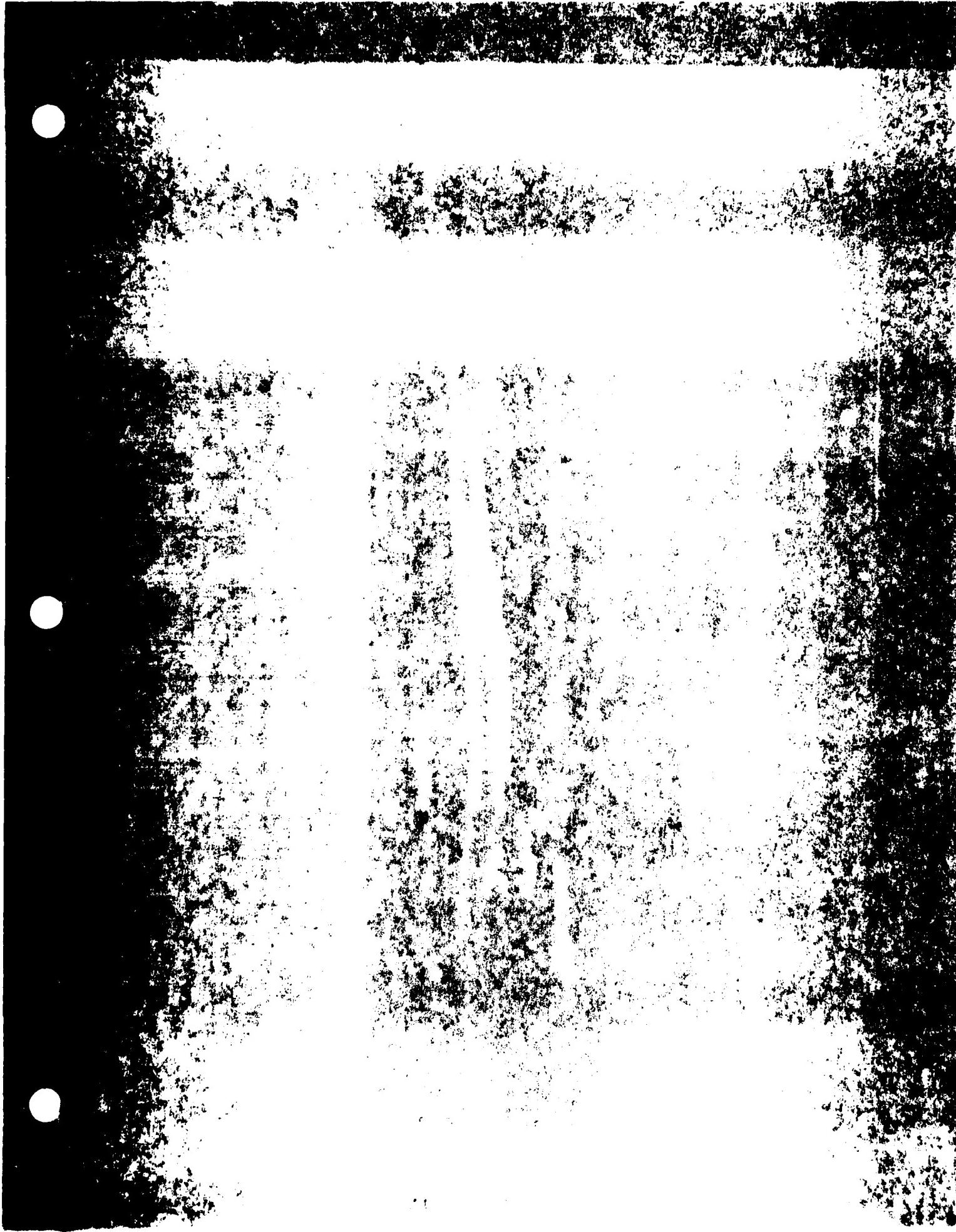
Plate No.

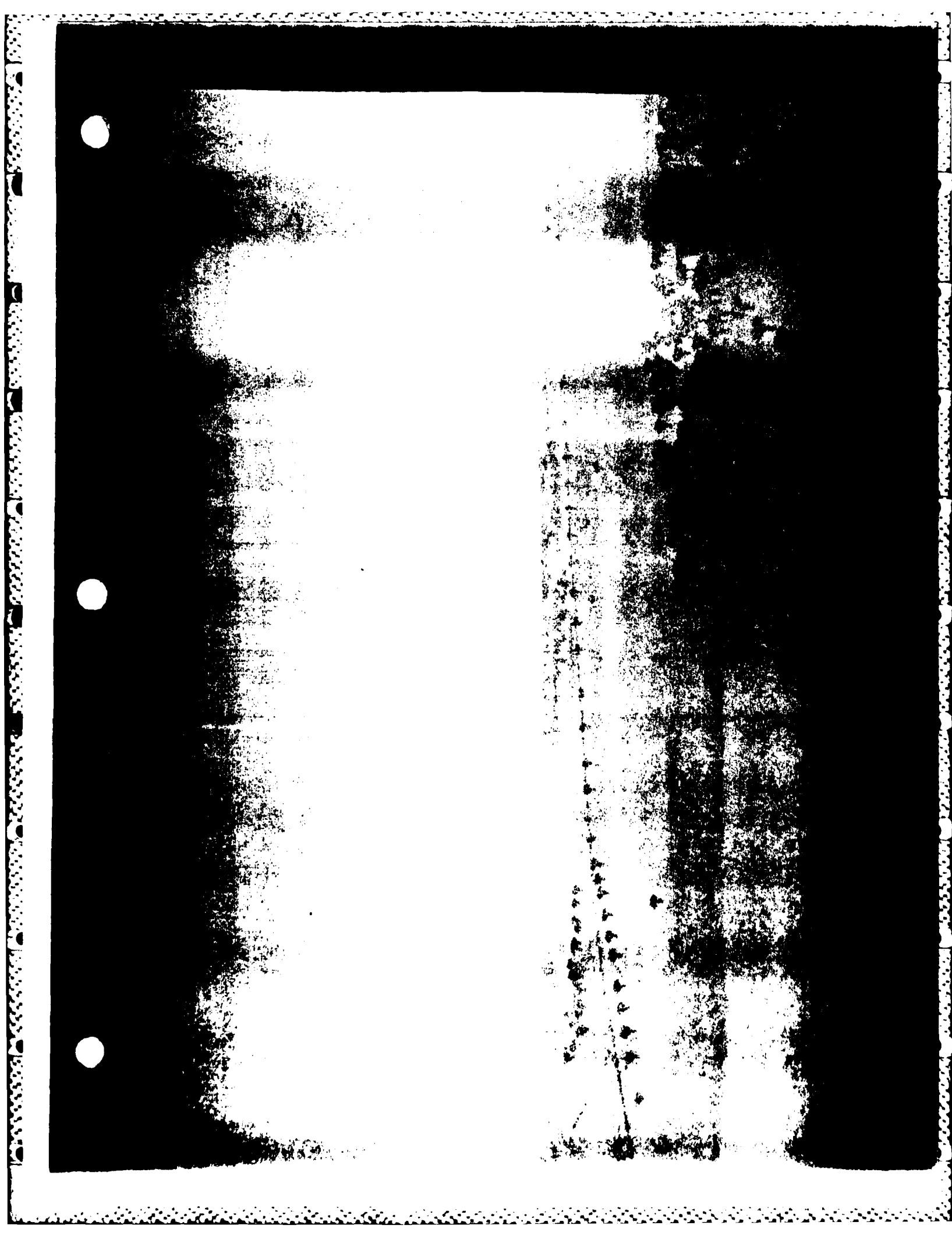
- I. Southern View of Mount Rainier utilizing Vertex Coloring
- II. Texture Mapping of the Lake Chelan Area of Washington State with an Even Pattern using the Mip Map Approach
- III. Texture Mapping of the Chelan Area of Washington State with an Uneven Pattern using the Mip Map Approach
- IV. Airfield with Haze Produced by Forcing Colors in the Background to Converge to Blue (Sky Color)
- V. Airfield at Night Produced by a Process Forcing Colors to Converge Toward Black, and utilizing Three-dimensional Pyramid Models to Produce Lights
- VI. Missiles with Shadows Created using Secondary Depth Buffers for Shadow Placement
- VII. Flying Saucer Generated using Quadric Surfaces
- VIII. Visual Representation of Kinematic Analysis on Trailing Edge Flaps
- IX. Airplane Interior Shot using Master Dimension Data which can be used to Facilitate Creation of Potential Aircraft Interiors





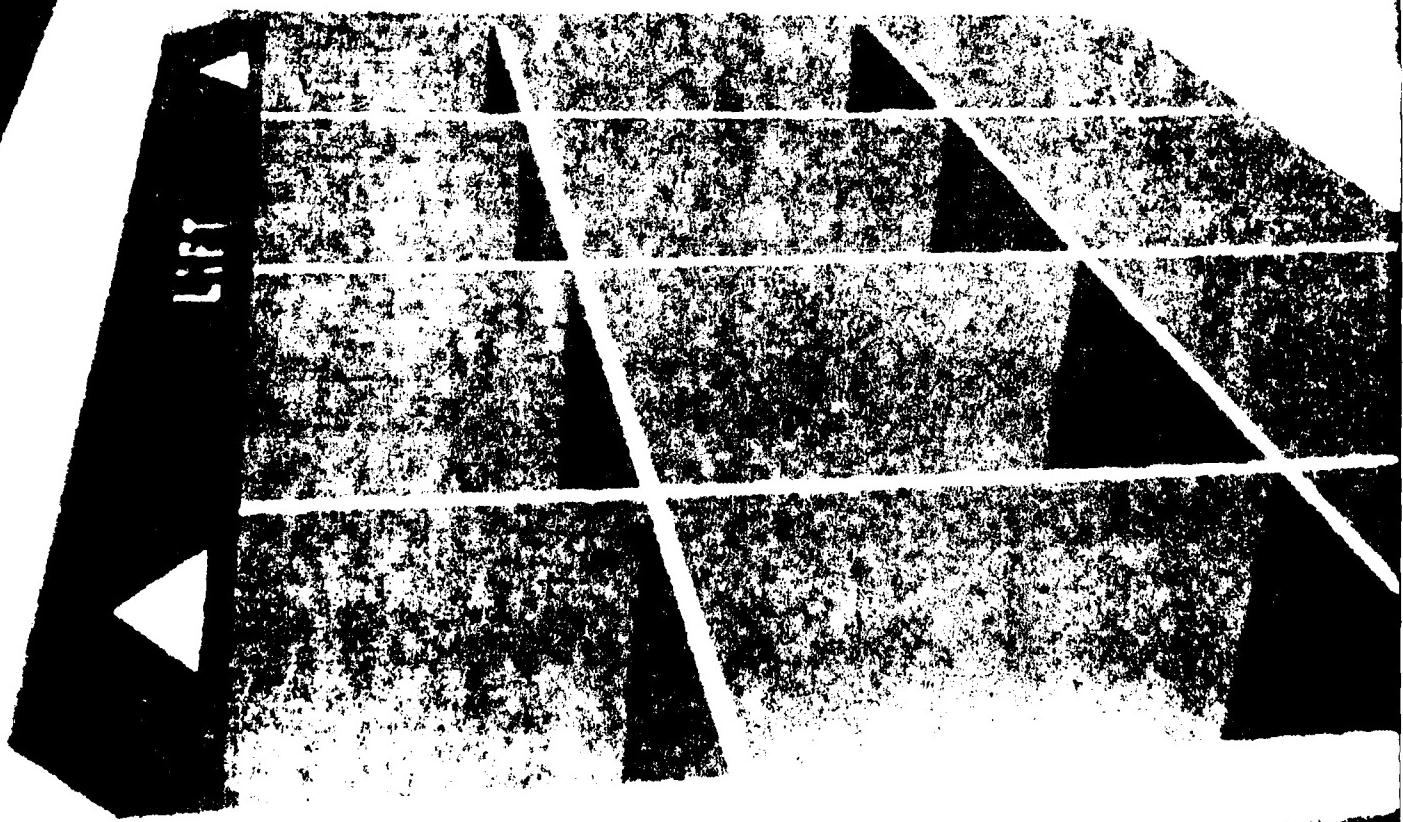


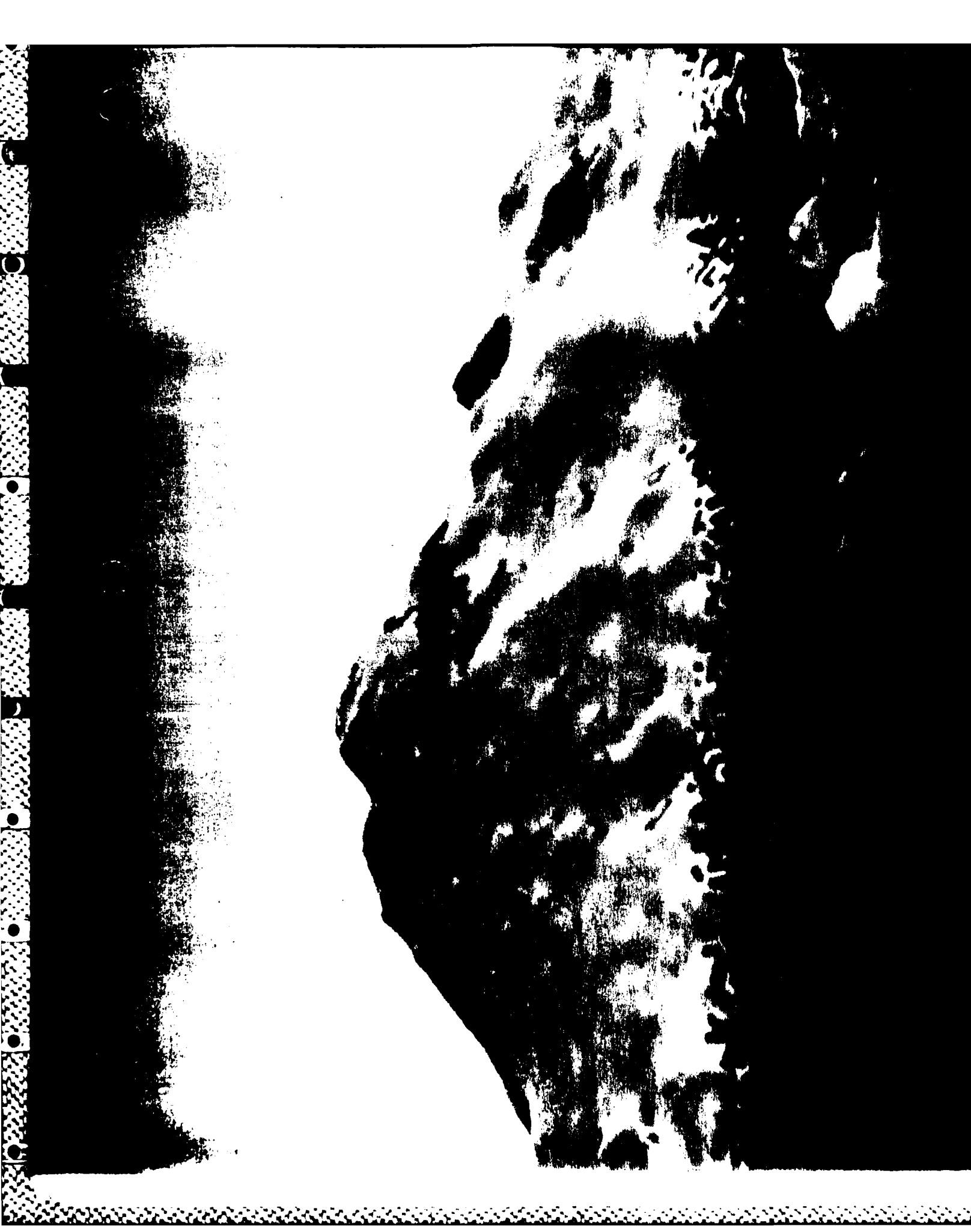












END

FILMED

10-84

DTIC